浙江大学

《数据库系统》
实验报告

作业名称	：	MiniSQL
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	孙建伶
助　　教	：	聂俊哲/石宇新

2024 年 6 月 8 日

- 实验名称 **7 LOCK MANAGER**
  - 实验目的
  - 实验环境
  - 实验流程
    - txn.h相关
      - 模块涉及到的C++类
        - C++11的std::thread
        - unordered_set
        - unordered_map
    - txn_manager事务管理器相关
      - 类内函数作用
        - Begin
        - Commit
        - Abort
        - GetTransaction
        - ReleaseLocks
      - 模块涉及到的C++类
        - std::atomic和std::mutex
        - std::mutex
        - std::atomic
        - shared_mutex
    - lock_manager相关
      - LockManager主类
      - LockRequest(Class)
      - LockRequestQueue(Class)
      - 模块涉及到的C++类
        - wait()成员函数
        - notify_all/notify_one
      - 实现函数
        - LockShared
        - LockExclusive
        - LockUpgrade
        - Unlock
        - LockPrepare
        - CheckAbort
        - AddEdge
        - RemoveEdge
        - HasCycle
        - DeleteNode
        - RunCycleDetection
        - DFS_Gwait
  - 遇到的问题及解决方法
  - 思考题

总结

实验名称 7 LOCK MANAGER

实验目的

> *本次实验中，你需要实现Lock Manager模块，从而实现并发的查询，Lock Manager负责追踪发放给事务的锁，并依据隔离级别适当地授予和释放shared(共享)和exclusive(独占)锁。*

实验环境

> *1. 操作系统：Windows 11 23H2*
>
> *2. 数据库管理系统：MiniSQL*
>
> *3. 工具：CLion 2023.3.3*
>
> *4. 工程管理：Git/GitHub*

实验流程

数据库系统中，事务管理器（Transaction Manager）是负责处理所有与事务相关操作的组件。它是维护数据库ACID属性（原子性、一致性、隔离性、持久性）的关键组件，确保了数据库系统中的事务能够安全、一致且高效地执行。事务管理器主要负责以下几个方面：

1. **事务的边界控制**：事务管理器负责定义事务的开始（BEGIN TRANSACTION）和结束（COMMIT 或 ROLLBACK）。当事务开始时，事务管理器会为其分配所需的资源，并追踪其状态。当事务成功完成时，事务管理器会执行提交操作，将所有更改永久写入数据库。如果事务遇到错误或者需要撤销，事务管理器将执行回滚操作，撤销所有更改。
2. **并发控制**：在允许多个事务同时运行的系统中，事务管理器使用并发控制机制（如锁、时间戳、版本号lsn等）来确保事务不会相互干扰，导致数据不一致。并发控制也包括实现数据库的隔离级别，防止并发事务产生冲突。
3. **恢复管理**：事务管理器还负责实现恢复机制，以保证在系统故障（如崩溃、电源中断）后数据库的一致性和持久性。这通常通过使用日志记录（Logging）和检查点（Checkpointing）等技术来完成。事务日志存储了所有对数据库所做的更改的记录，可以用于恢复操作。
4. **故障处理**：在检测到错误或异常时，事务管理器负责采取适当的行动，例如触发回滚来撤销事务的操作，或者在某些情况下，尝试恢复事务执行。

txn.h相关

该类详细介绍了事务对象应该包含的成员信息，同时也暗示，在事务进行上锁解锁时要进行事务信息方面的修改：

```
class Txn {
 public:
  explicit Txn(txn_id_t txn_id = INVALID_TXN_ID, IsolationLevel
iso_level = IsolationLevel::kRepeatedRead)
      : txn_id_(txn_id), iso_level_(iso_level),
thread_id_(std::this_thread::get_id()) {}

 private:
  txn_id_t txn_id_{INVALID_TXN_ID};
  IsolationLevel iso_level_{IsolationLevel::kRepeatedRead};
  TxnState state_{TxnState::kGrowing};
  std::thread::id thread_id_;
  std::unordered_set<RowId> shared_lock_set_;
  std::unordered_set<RowId> exclusive_lock_set_;
};
```

数据类型	变量名称	代表含义
txn_id_t	txn_id_	事务的唯一标识，
IsolationLevel	iso_level_	隔离等级：目前有三种`kReadUncommitted`，`kReadCommitted`，`kRepeatedRead`
TxnState	state_	在二阶段封锁协议下的事务状态(四种)：`kGrowing`，`kShrinking`，`kCommitted`，`kAborted`
std::thread::id	thread_id_	线程id，有关线程的成员函数可以查看`thread.h`
std::unordered_set	shared_lock_set_	(S锁)共享锁集合：以`unorder_set`形式储存，用法同`unorder_map`，其中内容是所占有的数据项的`Row_Id`
std::unordered_set	exclusive_lock_set_	(X锁)占有锁集合：以`unorder_set`形式储存，用法同`unorder_map`,其中内容是所占有的数据项的`Row_Id`

模块涉及到的C++类

C++11的std::thread

在C中已经有一个叫做pthread的东西来进行多线程编程，但是并不好用（如果你认为句柄、回调式编程很实用，那请当我没说），所以c++11标准库中出现了一个叫作std::thread的东西。

std::thread常用成员函数

- 构造&析构函数

函数	类别	作用
thread() noexcept	默认构造函数	创建一个线程，什么也不做
template <class Fn, class... Args> explicit thread(Fn&& fn, Args&&... args)	初始化构造函数	创建一个线程，以`args`为参数执行`fn`函数
thread(const thread&) = delete	复制构造函数	（已删除）
thread(thread&& x) noexcept	移动构造函数	构造一个与`x`相同的对象，会破坏`x`对象
~thread()	析构函数	析构对象

- 常用成员函数

函数	作用
void join()	等待线程结束并清理资源（会阻塞）
bool joinable()	返回线程是否可以执行join函数
void detach()	将线程与调用其的线程分离，彼此独立执行（此函数必须在线程创建时立即调用，且调用此函数会使其不能被join）
std::thread::id get_id()	获取线程id
thread& operator=(thread &&rhs)	见移动构造函数（如果对象是joinable的，那么会调用`std::terminate()`结果程序）

unordered_set

无序集合(`unordered_set`)是一种使用哈希表实现的无序关联容器，其中键被哈希到哈希表的索引位置，因此插入操作总是随机的。无序集合上的所有操作在平均情况下都具有常数时间复杂度O(1)，但在最坏情况下，时间复杂度可以达到线性时间O(n)，这取决于内部使用的哈希函数，但实际上它们表现非常出色，通常提供常数时间的查找操作。

无序集合可以包含任何类型的键 - 预定义或用户自定义数据结构，但所有键必须是唯一的。它的语法如下：

```
1 std::unordered_set<data_type> name;
```

常用方法

- `size()`和`empty()`:用于获取大小和集合是否为空
- `find()`:用于查找键
- `insert()`和`erase()`:用于插入和删除元素

如果集合中不存在某个键，`find()`函数会返回一个指向`end()`的迭代器，否则会返回指向键位置的迭代器。迭代器充当键值的指针，因此我们可以使用`*`运算符来解引用它们以获取键。

其他方法

FUNCTION NAME	FUNCTION DESCRIPTION
insert()	插入一个新元素
begin()/end()	返回一个迭代器，指向第一个元素/最后一个元素后的理论元素
count()	计算在无序集合容器中特定元素的出现次数
find()	搜索元素
clear()	清空所有元素
cbegin()/cend()	返回一个常量迭代器，指向第一个元素/最后一个元素后的理论元素
bucket_size()	返回无序集合中特定桶(bucket)中的元素总数(元素通过哈希函数映射到不同的桶中)
erase()	移除单个或某个范围内的一系列元素
size()	返回元素数量
swap()	交换两个无序集合容器的值
emplace()	在无序集合容器中插入元素

FUNCTION NAME	FUNCTION DESCRIPTION
max_size()	返回可以容纳的最大元素数量
empty()	检查无序集合容器是否为空
equal_range()	返回包括与给定值相等的所有元素的范围
hash_function()	用于获取容器所使用的哈希函数对象
reserve()	它用于请求容器预留足够的桶数，以容纳指定数量的元素
bucket()	返回特定元素的桶编号
bucket_count()	返回无序集合容器中的总桶数
load_factor()	用于获取当前容器的负载因子。负载因子:元素数量与桶数之比，用于衡量容器的填充程度
rehash()	设置容器的桶数以容纳一定数量的元素
max_load_factor()	获取或设置容器的最大负载因子
emplace_hint()	根据给定的提示位置(iterator)在容器中插入一个新元素
key_eq()	无序集合内部用于比较元素键值相等性的函数对象或谓词类型
max_bucket_count()	获取无序集合容器支持的最大桶数

unordered_map

无序映射(`unordered_map`)是一种关联容器，用于存储由键和映射值组成的元素。键值用于唯一标识元素，而映射值是与键相关联的内容。键和值都可以是任何预定义或用户定义的类型。简而言之，无序映射类似于一种字典类型的数据结构，可以在其中存储元素。

在内部，无序映射使用哈希表来实现，提供的键被哈希成哈希表的索引，因此数据结构的性能在很大程度上取决于哈希函数。但平均而言，从哈希表中搜索、插入和删除的成本是O(1)。

基本使用

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    unordered_map<string, int> umap;

    umap["GeeksforGeeks"] = 10;
    umap["Practice"] = 20;
```

```
    umap["Contribute"] = 30;

    for (auto x : umap)
        //Contribute=30 GeeksforGeeks=10 Practice=20
        cout << x.first << "=" << x.second << ' ';
}
```

方法

METHODS/FUNCTIONS	DESCRIPTION
at(K)	返回与元素作为键k相关的值的引用
begin()/end()	返回一个迭代器，指向第一个元素/最后一个元素后的理论元素
bucket(k)	返回键k所在的桶编号，即元素在映射中的位置。
bucket_count()	返回无序映射容器中的总桶数
bucket_size()	返回无序映射中每个桶中的元素数量
count()	计算具有给定键的元素在无序映射中出现的次数
equal_range(k)	返回包括与键k相等的所有元素的范围的边界
find()	搜索元素
empty()	检查无序映射容器是否为空
erase()	从无序映射容器中删除元素

C++11库还提供了用于查看内部使用的桶数、桶大小以及使用的哈希函数和各种哈希策略的函数(和`unordered_set`中类似)，但它们在实际应用中的用处较小。我们可以使用迭代器遍历无序映射中的所有元素。

txn_manager事务管理器相关

在本次实验中，我们提供的`TxnManager`主要负责事务的边界控制、并发控制、故障处理。出于实现复杂度的考虑，同时为了避免各模块耦合太强，前面模块的问题导致后面模块完全无法完成，我们将`TxnManager`模块单独拆了出来。`TxnManager`的代码已经实现好了，支持`Begin()`、`Commit()`、`Abort()`等方法。因为`TxnManager`模块独立，我们在`Commit()`、`Abort()`方法中不需要做其他事情（本来需要维护事务中的写、删除集合，结合`Recovery`模块回滚）。同时我们提供了`Txn`类,里面通过参数控制事务的隔离级别：

- READ_UNCOMMITED
- READ_COMMITTED
- REPEATABLE_READ

Lock Manager 负责检查事务的隔离级别，任何失败的锁操作都将导致事务中止，并同时抛出异常，此时`TxnManager`将捕获该异常并回滚。

```
class TxnManager {
 public:
  explicit TxnManager(LockManager *lock_mgr);
 private:
  LockManager *lock_mgr_{nullptr};
  std::atomic<txn_id_t> next_txn_id_{0};
  /** The transaction map is a global list of all the running transactions in the system. */
  std::unordered_map<txn_id_t, Txn *> txn_map_{};
  std::shared_mutex rw_latch_{};
};
```

数据类型	变量名称	代表含义
LockManager *	lock_mgr_	指向当前数据库的锁处理器，可以通过锁处理器来处理真正底层锁的上锁解除
std::atomic<txn_id_t>	next_txn_id_	原子变量可以方便创建事务时不会少数漏数事务。代表含义是下一个创建事务的id号，在每次创建后会+1
std::unordered_map<txn_id_t, Txn *>	txn_map_	通过事务号来获取实际事务的映射，方法操作较常用的是`[]`、`find()`。主要是找到对应迭代器来进行事务的修改
std::shared_mutex	rw_latch_	`shard_mutex`类型与我们要实现的共享锁、排他锁很相像，此处作用不明，猜测是读写状态存储

这一部分文件将事务管理独立了出来作锁操作的设计与验证，与之前的模块相独立

类内函数作用

Begin

```
Txn *TxnManager::Begin(Txn *txn, IsolationLevel isolationLevel) {
  if (nullptr == txn) {
    txn = new Txn(next_txn_id_++, isolationLevel);
  }
```

```
  std::unique_lock<std::shared_mutex> lock(rw_latch_);
  txn_map_[txn->GetTxnId()] = txn;
  return txn;
}
```

函数作用:

- 实现事务的新建,事务号是根据`next_txn_id_`的递增实现的
- 请求对 `rw_latch_` 的独占访问。如果 `rw_latch_` 已被其他线程锁定，当前线程将被阻塞，直到锁可用
- 添加事务到 `txn_map_`中,供之后函数调用,标识事务的存在

Commit

```
void TxnManager::Commit(Txn *txn) {
  // change state
  txn->SetState(TxnState::kCommitted);
  // release all locks
  ReleaseLocks(txn);
}
```

函数作用:

- 设置事务状态为提交
- 释放事务所占有的所有锁

Abort

```
void TxnManager::Abort(Txn *txn) {
  // change state
  txn->SetState(TxnState::kAborted);
  // release all locks
  ReleaseLocks(txn);
}
```

函数作用:

- 设置事务状态为取消
- 释放事务所占有的所有锁

GetTransaction

```
Txn *TxnManager::GetTransaction(txn_id_t txn_id) {
  std::shared_lock<std::shared_mutex> lock(rw_latch_);
  auto iter = txn_map_.find(txn_id);
  if (iter != txn_map_.end()) {
    return iter->second;
  }
  return nullptr;
}
```

函数作用:

- 请求对 `rw_latch_` 的独占访问。如果 `rw_latch_` 已被其他线程锁定，当前线程将被阻塞，直到锁可用
- 查找对应事务号的事务,如果不存在则返回空指针

ReleaseLocks

```
void TxnManager::ReleaseLocks(Txn *txn) {
  std::unordered_set<RowId> lock_set;
  for (auto o : txn->GetExclusiveLockSet()) {
    lock_set.emplace(o);
  }
  for (auto o : txn->GetSharedLockSet()) {
    lock_set.emplace(o);
  }
  for (auto rid : lock_set) {
    lock_mgr_->Unlock(txn, rid);
  }
}
```

函数作用:

- 请求对 `rw_latch_` 的独占访问。如果 `rw_latch_` 已被其他线程锁定，当前线程将被阻塞，直到锁可用
- 对于事务本身而言，清空自身的存储的锁的集合
- 对于LM而言，执行`Unlock`清空锁集合

模块涉及到的C++类

std::atomic和std::mutex

```
#include <iostream>
#include <thread>
using namespace std;
int n = 0;
void count10000() {
    for (int i = 1; i <= 10000; i++)
        n++;
}
int main() {
    thread th[100];
    // 这里偷了一下懒，用了c++11的foreach结构
    for (thread &x : th)
        x = thread(count10000);
    for (thread &x : th)
        x.join();
    cout << n << endl;
    return 0;
}
1234567891011121314151617181920
```

2次输出结果分别是：

```
991164
996417
12
```

我们的输出结果应该是1000000，可是为什么实际输出结果比1000000小呢？

在上文我们分析过多线程的执行顺序——同时进行、无次序，所以这样就会导致一个问题：多个线程进行时，如果它们同时操作同一个变量，那么肯定会出错。为了应对这种情况，c++11中出现了`std::atomic`和`std::mutex`。

std::mutex

`std::mutex`是 C++11 中最基本的互斥量，一个线程将mutex锁住时，其它的线程就不能操作mutex，直到这个线程将mutex解锁。

mutex的常用成员函数（这里用`mutex`代指`对象`）

函数	作用
void lock()	将mutex上锁。 如果mutex已经被其它线程上锁， 那么会阻塞，直到解锁；如果mutex已经被同一个线程锁住， 那么会产生死锁。
void unlock()	解锁mutex，释放其所有权。 如果有线程因为调用lock()不能上锁而被阻塞，则调用此函数会将mutex的主动权随机交给其中一个线程； 如果mutex不是被此线程上锁，那么会引发未定义的异常。
bool try_lock()	尝试将mutex上锁。 如果mutex未被上锁，则将其上锁并返回true； 如果mutex已被锁则返回false。

std::atomic

> *原子操作是最小的且不可并行化的操作。*

这就意味着即使是多线程，也要像同步进行一样**同步操作**atomic对象，从而省去了mutex上锁、解锁的时间消耗。

std::atomic常用成员函数

函数	类型	作用
atomic() noexcept = default	默认构造函数	构造一个atomic对象（未初始化，可通过atomic_init进行初始化）
constexpr atomic(T val) noexcept	初始化构造函数	构造一个atomic对象，用`val`的值来初始化
atomic(const atomic&) = delete	复制构造函数	（已删除）

shared_mutex

C++17起。`shared_mutex`类是一个同步原语，可用于保护共享数据不被多个线程同时访问。与便于独占访问的其他互斥类型不同，`shared_mutex` 拥有二个访问级别：

- 共享 - 多个线程能共享同一互斥的所有权；
- 独占性 - 仅一个线程能占有互斥。

1）若一个线程已经通过lock或try_lock获取独占锁（写锁），则无其他线程能获取该锁（包括共享的）。尝试获得读锁的线程也会被阻塞。
2）仅当任何线程均未获取独占性锁时，共享锁（读锁）才能被多个线程获取（通过lock_shared 、try_lock_shared ）。
3）在一个线程内，同一时刻只能获取一个锁（共享或独占性）。

成员函数主要包含两大类：**排他性锁定（写锁）和共享锁定（读锁）。**

lock_manager相关

Lock Manager的基本思想是它维护当前活动事务持有的锁。事务在访问数据项之前向 LM 发出锁请求，LM 来决定是否将锁授予该事务，或者是否阻塞该事务或中止事务。LM里定义了两个内部类：`LockRequest` and `LockRequestQueue`.

在你的实现当中，整个数据库系统会存在一个全局的 LM 结构。每当一条事务需要去访问一条数据记录时，借助该全局的LM去获取数据记录上的锁。条件变量可用于阻塞等待直到它们的锁请求得到满足的事务。本次实验中，同学们实现的LM需要支持三种不同的隔离级别。

Note：

- 在锁管理器需要使用死锁检测时，我们建议首先实现一个不包含任何死锁处理的锁管理器，然后在确认其在没有死锁发生时能够正确地进行锁定和解锁后，再添加检测机制。
- 虽然通过确保严格两阶段锁（strict two phase lock）可以实现某些隔离级别，但本次实验的锁管理器实现只需确保两阶段锁的特性。严格两阶段锁的概念将通过执行器和事务管理器中的逻辑来实现。具体需要查看其中的 `Commit` 和 `Abort` 方法。
- 还需要跟踪事务所获取的共享/独占锁，使用 `shared_lock_set_` 和 `exclusive_lock_set_`，这样当 `TransactionManager` 想要提交/中止事务时，LM能够适当地释放它们。

本次实验实现的锁管理器应该在后台运行死锁检测，以中止阻塞事务。更准确地说，这意味着一个后台线程应该定期即时构建一个等待图，并打破任何循环。需要实现并用于循环检测以及测试的API如下：

- `AddEdge(txn_id_t t1, txn_id_t t2)`：在图中从t1到t2添加一条边。如果该边已存在，则无需进行任何操作。
- `RemoveEdge(txn_id_t t1, txn_id_t t2)`：从图中移除t1到t2的边。如果没有这样的边存在，则无需进行任何操作。

- `HasCycle(txn_id_t& txn_id)`：使用深度优先搜索(DFS)算法寻找循环。如果找到循环，`HasCycle`应该将循环中最早事务的id存储在`txn_id`中并返回true。该函数应该返回它找到的第一个循环。如果图中没有循环，`HasCycle`应该返回false。
- `GetEdgeList()`：返回一个元组列表，代表图中的边。一对(t1,t2)对应于从t1到t2的一条边。
- `RunCycleDetection()`：包含在后台运行循环检测的框架代码。需要在此实现循环检测逻辑。

实现完成后，你的代码需要通过`lock_manager_test.cpp`中的所有测试用例。

Note:

- 后台线程应该在每次唤醒时即时构建图表，而不是维护一个图表。等待图应该在每次线程唤醒时构建和销毁。
- 实验中的DFS循环检测算法**必须**是确定性的。为了做到这一点，必须始终选择首先探索最低的事务ID。这意味着在选择从哪个**未探索**的节点运行DFS时，始终选择具有最低事务ID的节点。这也意味着在探索邻居时，按从最低到最高的顺序探索它们。
- 当发现循环时，应该通过将该事务的状态设置为ABORTED来中止**最年轻**的事务以打破循环。
- 当检测线程唤醒时，它负责打破存在的**所有**循环。如果你遵循上述要求，你将总是以**确定性**的顺序找到循环。这也意味着当你构建图时，你**不应该**为已中止的事务添加节点或向已中止的事务绘制边。
- 等待图是一个**有向**图。当一个事务在等待另一个事务时，等待图会画出边。如果多个事务持有一个**共享**锁，一个单独的事务可能会等待多个事务。
- 当一个事务被中止时，确保将事务的状态设置为`ABORTED`并在您的锁管理器中抛出一个异常。事务管理器将负责明确的中止和回滚更改。一个等待锁的事务可能会被后台循环检测线程中止。您必须有一种方法通知等待的事务它们已被中止。

LockManager主类

```
class LockManager {
public:
  enum class LockMode { kNone, kShared, kExclusive };
private:
  /** Lock table for lock requests. */
  std::unordered_map<RowId, LockRequestQueue> lock_table_{};
  std::mutex latch_{};

```

```
  /** Waits-for graph representation. */
  std::unordered_map<txn_id_t, std::set<txn_id_t>> waits_for_{};
  std::unordered_set<txn_id_t> visited_set_{};
  std::stack<txn_id_t> visited_path_{};
  txn_id_t revisited_node_{INVALID_TXN_ID};
  std::atomic<bool> enable_cycle_detection_{false};
  std::chrono::milliseconds cycle_detection_interval_{100};
  TxnManager *txn_mgr_{nullptr};
};
```

数据类型	变量名称	代表含义
std::unordered_map<RowId, LockRequestQueue>	lock_table_	这个包含了对应数据项的上锁请求队列
std::mutex	mutex latch_	与`shared_mutex`不同，这里的数据类型只支持单纯的上锁解锁，没有隔壁的共享排他之分，用法一般是`lock`、`unlock`,
std::unordered_map<txn_id_t, std::set<txn_id_t>>	waits_for_	等待图存储单元，与DS课程教学不同，这里没有使用邻接链表来存储图，而是继续使用我们这一工程常使用的`unorder_map`数据类型。其中等待图的边是指等待关系$T_1 \rightarrow T_2$指T_1等待T_2释放锁。
std::unordered_set<txn_id_t>	visited_set_	利用`unordered_set`存储所有遇到过的事务，事务以事务号标识，可以通过find()来判断有没有经过该点。
std::stack<txn_id_t>	visited_path_	以栈存储经过的点的集合，这里与上面不同的地方是，栈能够真切反映访问的顺序。
txn_id_t	revisited_node_	字面意思是再次遇到的节点，其实就是这个闭环中你最开始搜索的结果，也就是DFS返回的结果。

数据类型	变量名称	代表含义
std::atomic	enable_cycle_detection_	原子变量，即最小的能够并行修改的变量。翻译一下就是这个变量能够实时反映现在的值，而不会受线程阻塞影响。
std::chrono::milliseconds	cycle_detection_interval_	时间常数，用来即时生成等待图使用，时间常数越小，刷新频率越快，死锁检测处理更快，但是也会带来进程的大量调用，影响总执行速度。
TxnManager *	txn_mgr_	事务管理器：能够统筹所有事务的管理器，如果有对事务的修改，请认准该管理器。

在网上查找`std::lock_guard`用于管理`std::mutex`，`std::unique_lock`与`std::shared_lock`管理`std::shared_mutex`，但是在条件变量搭配使用中只能使用`std::unique_lock<std::mutex>`，这样才能达到上锁的效果

LockRequest(Class)

```
 class LockRequest {
   public:
    LockRequest(txn_id_t txn_id, LockMode lock_mode)
        : txn_id_(txn_id), lock_mode_(lock_mode),
granted_(LockMode::kNone) {}

    txn_id_t txn_id_{0};
    LockMode lock_mode_{LockMode::kShared};
    LockMode granted_{LockMode::kNone};
  };

```

数据类型	变量名称	代表含义
txn_id_t	txn_id_	标识所在事务的id
LockMode	lock_mode_	锁的类型，分为共享锁、排他锁、未知节点(应该用不上)

数据类型	变量名称	代表含义
LockMode	granted_	已经获得锁的类型共享锁、排他锁、未知节点(就是指没有获得任何锁)

LockRequest：此类代表由事务（`txn_id`）发出的锁请求。它包含以下成员：

- `txn_id_`：发出请求的事务的标识符。
- `lock_mode_`：请求的锁类型（例如，共享或排他）。
- `granted_`：已授予事务的锁类型。

构造函数使用给定的`txn_id`和`lock_mode`初始化这些成员，默认将`granted_`设置为`LockMode::kNone`

该类型是描述一个事务对某个数据项(具体数据标识row_id在最高类lock_manager中的map关联first中)提出的上锁请求

[Tips]为了设计层级逻辑，这里的数据项信息来源于顶层类`lock_manager`中`lock_table_`相关联项中first值。

LockRequestQueue(Class)

```
class LockRequestQueue {
public:
    using ReqListType = std::list<LockRequest>;
public:
    ReqListType req_list_{};
    std::unordered_map<txn_id_t, ReqListType::iterator>
req_list_iter_map_{};
    std::condition_variable cv_{};
    bool is_writing_{false};
    bool is_upgrading_{false};
    int32_t sharing_cnt_{0};
};
```

数据类型	变量名称	代表含义
ReqListType	req_list_	描述当前数据项存在的上锁请求，使用list来存储锁的请求

数据类型	变量名称	代表含义
std::unordered_map<txn_id_t, ReqListType::iterator>	req_list_iter_map_	已知对应的数据项，跟踪列表中每个请求的迭代器(即跟踪每个请求，`key`值为请求事务的`txn_id`)
std::condition_variable	cv_	条件变量，负责进程的等待和接受`Lamda`函数捕获以达到函数暂停的作用，主要用法是`wait()`
bool	is_writing_	标识该数据项是否有事务在写，通常和数据项上的排他锁数量一致，但是该变量的存在使得我们可以不必使用排他锁的存在与否标识数据项的锁定。但是话又说回来，你也可以用排他锁的存在与否把这个变量代替掉
bool	is_upgrading_	标识现在的进程中是否有锁升级，推测是锁升级中不能进行任何数据改写的事务
int32_t	sharing_cnt_	表明该数据项有多少共享锁占用，一般上锁要+1,释放记得删

LockRequestQueue：此类管理一个锁请求队列，并提供操作它的方法。它使用一个列表（`req_list_`）存储请求，并使用一个`unordered_map(req_list_iter_map_)`跟踪列表中每个请求的迭代器。它还包括一个条件变量（`cv_`）用于同步目的，以及一些标志来管理并发访问：

- `is_writing_`：指示当前是否持有排他性写锁。
- `is_upgrading_`：指示是否正在进行锁升级。
- `sharing_cnt_`：持有共享锁的事务数量的整数计数。

该类提供以下方法：

- `EmplaceLockRequest()`：将新的锁请求添加到队列前端，并在map中存储其迭代器。
- `EraseLockRequest()`：根据`txn_id`从队列和map中移除锁请求。如果成功返回`true`，否则返回`false`。
- `GetLockRequestIter()`：根据`txn_id`检索队列中特定锁请求的迭代器。

[Tips]为了设计层级逻辑，这里的数据项信息来源于顶层类`lock_manager`中`lock_table_`相关联项中first值

模块涉及到的C++类

wait()成员函数

函数声明如下：

```
void wait(std::unique_lock<std::mutex>& lock);
//Predicate 谓词函数，可以普通函数或者lambda表达式
template<class Predicate>
void wait(std::unique_lock<std::mutex>& lock, Predicate pred);
```

`wait`导致当前线程阻塞直至条件变量被通知，或虚假唤醒发生，可选地循环直至满足某谓词。

notify_all/notify_one

notify函数声明如下：

```
void notify_one() noexcept;
```

若任何线程在 *this 上等待，则调用 notify_one 会解阻塞(唤醒)等待线程之一。

```
void notify_all() noexcept;
```

若任何线程在 *this 上等待，则解阻塞（唤醒)全部等待线程

实现函数

LockShared

`LockShared(Txn,RID)`：事务txn请求获取id为rid的数据记录上的共享锁。当请求需要等待时，该函数被阻塞（使用cv_.wait），请求通过后返回True

函数实现：

```
bool LockManager::LockShared(Txn *txn, const RowId &rid) {
  auto& req = lock_table_[rid];
  if(req.req_list_iter_map_.find(txn-
>GetTxnId())!=req.req_list_iter_map_.end()){
```

```
    if(req.GetLockRequestIter(txn->GetTxnId())->granted_==LockMode::kShared){
      return true;
    }
  }
  unique_lock<mutex> lock(latch_);
  if(txn->GetIsolationLevel() == IsolationLevel::kReadUncommitted){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockSharedOnReadUncommitted);
  }
  if(txn->GetState() == TxnState::kShrinking){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockOnShrinking);
  }
  LockPrepare(txn,rid);
  req.EmplaceLockRequest(txn->GetTxnId(),LockMode::kShared);
  if(req.is_writing_ == true){
    req.cv_.wait(lock,[txn,&req]() -> bool{return txn->GetState()==TxnState::kAborted||!req.is_writing_;});
  }
  CheckAbort(txn,req);
  //modify txn
  txn->GetSharedLockSet().emplace(rid);
  //modify lck_manager
  req.sharing_cnt_++;
  //modify request
  req.GetLockRequestIter(txn->GetTxnId())->granted_ = LockMode::kShared;
  return true;
}
```

1. 检查是否已有请求：

   首先，函数通过`lock_table_`查找是否有当前事务的锁请求。如果事务ID在`req_list_iter_map_`中存在，说明事务已经请求了锁。

2. 检查锁是否已授予：

   如果事务的请求存在，函数检查该请求是否已经被授予了共享锁（`LockMode::kShared`）。如果是，函数直接返回`true`。

3. 获取互斥锁：

   使用`unique_lock`来获取`latch_`互斥锁，确保线程安全。

4. 检查事务隔离级别：

   如果事务的隔离级别是`kReadUncommitted`，事务状态将被设置为`kAborted`，并抛出`TxnAbortException`异常。

5. 检查事务状态：

   如果事务状态是`kShrinking`，表示事务正在处于二阶段的下降阶段，这将导致事务被中止，并抛出异常。

6. 准备锁请求：

   调用`LockPrepare`函数来准备锁请求。

7. 创建锁请求：

   在`req`对象中为当前事务创建一个新的共享锁请求。

8. 等待锁请求：

   如果`req`对象中存在写锁请求，当前事务将等待直到它被中止或写锁请求不再存在。

9. 检查事务是否中止：

   调用`CheckAbort`函数检查事务是否已经被中止。

10. 修改事务状态：

    如果事务没有被中止，将`rid`添加到事务的共享锁集合中。

11. 增加共享计数：

    在`req`对象中增加共享锁的计数。

12. 授予共享锁：

    将事务的锁请求状态设置为已授予共享锁。

13. 返回结果：

    函数返回`true`，表示共享锁请求成功。

LockExclusive

`LockExclusive(Txn,RID)`: 事务txn请求获取id为rid的数据记录上的独占锁。当请求需要等待时，该函数被阻塞，请求通过后返回True

函数实现：

```
1  bool LockManager::LockExclusive(Txn *txn, const RowId &rid) {
2    auto& req = lock_table_[rid];
```

```
  if(req.req_list_iter_map_.find(txn->GetTxnId())!=req.req_list_iter_map_.end()){
    if(req.GetLockRequestIter(txn->GetTxnId())->granted_==LockMode::kExclusive){
      return true;
    }
  }
  unique_lock<mutex> lock_latch(latch_);
  if(txn->GetIsolationLevel() == IsolationLevel::kReadUncommitted){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockSharedOnReadUncommitted);
  }
  if(txn->GetState() == TxnState::kShrinking){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockOnShrinking);
  }
  LockPrepare(txn,rid);
  req.EmplaceLockRequest(txn->GetTxnId(),LockMode::kExclusive);
  if(req.is_writing_ == true||req.sharing_cnt_ !=0){
    req.cv_.wait(lock_latch,[txn,&req]() -> bool{
      return txn->GetState()==TxnState::kAborted||(!req.is_writing_&&req.sharing_cnt_==0);
    });
  }
  CheckAbort(txn,req);
  //modify txn
  txn->GetExclusiveLockSet().emplace(rid);
  //modify lck_manager
  req.is_writing_ = true;
  //modify request
  req.GetLockRequestIter(txn->GetTxnId())->lock_mode_ = LockMode::kExclusive;
  req.GetLockRequestIter(txn->GetTxnId())->granted_ = LockMode::kExclusive;
  return true;
}

```

1. 检查现有请求：

   函数首先检查事务是否已经对给定的`RowId`有锁请求。如果存在，并且该请求已经被授予了排他锁，则函数直接返回`true`。

2. 获取互斥锁：

   使用`unique_lock`来获取名为`latch_`的互斥锁，确保在修改锁表时的线程安全。

3. 检查事务隔离级别：

   如果事务的隔离级别是`kReadUncommitted`，事务状态将被设置为`kAborted`，并抛出`TxnAbortException`异常。

4. 检查事务状态：

   如果事务状态是`kShrinking`，表示事务位于二阶段封锁协议的下降阶段，这将导致事务被中止，并抛出异常。

5. 准备锁请求：

   调用`LockPrepare`函数来准备锁请求。

6. 创建排他锁请求：

   在`req`对象中为当前事务创建一个新的排他锁请求。

7. 等待锁请求：

   如果存在写锁（`is_writing_`为`true`）或者有其他事务持有共享锁（`sharing_cnt_`不为0），当前事务将等待直到它被中止，或者写锁不存在且没有其他共享锁。

8. 检查事务是否中止：

   调用`CheckAbort`函数来检查事务是否已经被中止。

9. 修改事务状态：

   如果事务没有被中止，将`rid`添加到事务的排他锁集合中。

10. 修改锁请求：

    将事务的锁模式设置为排他锁，并将锁请求的状态更新为已授予排他锁。

11. 修改锁管理器状态：

    将`req`对象的`is_writing_`标志设置为`true`，表示现在有事务持有排他锁。

12. 返回结果：

    函数返回`true`，表示排他锁请求成功。

LockUpgrade

`LockUpgrad(Txn,RID)`:事务txn请求更新id为rid的数据记录上的独占锁，当请求需要等待时，该函数被阻塞，请求通过后返回True

函数实现：

```
bool LockManager::LockUpgrade(Txn *txn, const RowId &rid) {
  auto& req = lock_table_[rid];
  if(req.req_list_iter_map_.find(txn->GetTxnId())!=req.req_list_iter_map_.end()){
    if(req.GetLockRequestIter(txn->GetTxnId())->granted_==LockMode::kExclusive){
      return true;
    }
  }
  unique_lock<mutex> lock_latch(latch_);
  if(txn->GetState() == TxnState::kShrinking){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockOnShrinking);
  }

  if(req.is_upgrading_ == true){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kUpgradeConflict);
  }
req.GetLockRequestIter(txn->GetTxnId())->lock_mode_=LockMode::kExclusive;
  if(req.is_writing_ == true||req.sharing_cnt_ >1){
    req.is_upgrading_ = true;
    req.cv_.wait(lock_latch,[txn,&req]() -> bool{
      return txn->GetState()==TxnState::kAborted||(!req.is_writing_&&req.sharing_cnt_==1);
    });
  }
  if(txn->GetState() == TxnState::kAborted){
    req.is_upgrading_ = false;
  }
  CheckAbort(txn,req);
  //modify txn
```

```
30    txn->GetSharedLockSet().erase(rid);
31    txn->GetExclusiveLockSet().emplace(rid);
32    //modify lck_manager
33    req.is_writing_= true;
34    req.is_upgrading_ = false;
35    req.sharing_cnt_--;
36    req.is_writing_ = true;
37    //modify request
38    req.GetLockRequestIter(txn->GetTxnId())->granted_ =
    LockMode::kExclusive;
39    return true;
40  }
```

1. 检查现有请求：

   函数首先检查事务是否已经对给定的`RowId`有锁请求。如果存在，并且该请求已经被授予了排他锁，则函数直接返回`true`。

2. 获取互斥锁：

   使用`unique_lock`来获取名为`latch_`的互斥锁，确保在修改锁表时的线程安全。

3. 检查事务状态：

   如果事务状态是`kShrinking`，表示事务正在两阶段封锁协议的下降阶段，这将导致事务被中止，并抛出`TxnAbortException`异常。

4. 检查是否正在升级：

   如果`req`对象的`is_upgrading_`标志为`true`，表示其他事务正在尝试升级锁，当前事务将被中止，并抛出异常。

5. 修改锁请求模式：

   将当前事务的锁请求模式设置为排他锁。

6. 等待锁升级：

   如果存在写锁（`is_writing_`为`true`）或者共享计数大于1（`sharing_cnt_ > 1`），当前事务将设置`is_upgrading_`为`true`并等待直到它被中止，或者没有其他写锁且只有一个共享锁。

7. 检查事务是否中止：

   如果事务状态变为`kAborted`，在退出等待之前将`is_upgrading_`设置回`false`。

8. 检查事务是否中止：

   再次检查事务是否已经被中止，如果是，则不需要进行后续操作。

9. 检查并处理中止：

调用`CheckAbort`函数来检查事务是否已经被中止。

10. 修改事务锁集合：

    从事务的共享锁集合中移除`rid`，并将`rid`添加到事务的排他锁集合中。

11. 修改锁管理器状态：

    将`req`对象的`is_writing_`设置为`true`，表示现在有事务持有排他锁，并将`is_upgrading_`设置为`false`，表示锁升级完成。同时减少共享计数`sharing_cnt_`。

12. 授予排他锁：

    将事务的锁请求状态更新为已授予排他锁。

13. 返回结果：

    函数返回`true`，表示锁升级成功。

Unlock

`Unlock(Txn,RID)`：释放事物txn在rid数据记录上的锁。注意维护事务的状态，例如该操作中事务的状态可能会从 `GROWING` 阶段变为 `SHRINKING` 阶段（提示：查看 `transaction.h` 中的方法）。此外，当需要某种方式来通知那些等待中的事务，我们可以使用`notify_all()`方法

函数实现：

```
bool LockManager::Unlock(Txn *txn, const RowId &rid) {
  unique_lock<mutex> lock_latch(latch_);
  auto& req = lock_table_[rid];
  if(req.req_list_iter_map_.find(txn->GetTxnId()) ==
req.req_list_iter_map_.end()){
    return false;
  }
  if(txn->GetState() == TxnState::kGrowing&&txn-
>GetIsolationLevel() != IsolationLevel::kReadCommitted){
    txn->SetState(TxnState::kShrinking);
  }
  //modify request
  switch(req.GetLockRequestIter(txn->GetTxnId())->granted_){
    case LockMode::kShared:
      req.sharing_cnt_--;
      break;
    case LockMode::kExclusive:
      req.is_writing_ = false;
```

```
17        break;
18      default:
19        break;
20    }
21    txn->GetSharedLockSet().erase(rid);
22    txn->GetExclusiveLockSet().erase(rid);
23    req.GetLockRequestIter(txn->GetTxnId())->granted_ =
LockMode::kNone;
24    req.cv_.notify_all();
25    return true;
26  }
```

1. 获取互斥锁：

   使用`unique_lock`来获取名为`latch_`的互斥锁，确保在修改锁表时的线程安全。

2. 检查事务请求存在：

   函数首先检查事务是否对给定的`RowId`有锁请求。如果没有，函数直接返回`false`。

3. 检查事务状态：

   如果事务状态是`kGrowing`且隔离级别不是`kReadCommitted`，则将事务状态更新为`kShrinking`。

4. 修改锁请求：

   根据事务持有的锁类型（共享或排他），执行不同的操作：

   - 如果是共享锁（`LockMode::kShared`），减少共享计数`sharing_cnt_`。
   - 如果是排他锁（`LockMode::kExclusive`），将`is_writing_`标志设置为`false`。

5. 从事务锁集合中移除：

   从事务的共享锁集合和排他锁集合中移除对应的`RowId`。

6. 重置锁请求状态：

   将事务的锁请求状态设置为无锁（`LockMode::kNone`）。

7. 通知等待的事务：

   使用条件变量`cv_`的`notify_all`方法通知所有等待该锁的事务，因为锁已经被释放。

8. 返回结果：

   函数返回`true`，表示锁已成功释放。

LockPrepare

`LockPrepare(Txn,RID)`：检测txn的state是否符合预期，并在`lock_table_`里创建rid和对应的队列

函数实现：

```
void LockManager::LockPrepare(Txn *txn, const RowId &rid) {
  if(txn->GetState()!= TxnState::kShrinking){
    if(lock_table_.find(rid) == lock_table_.end()){
      lock_table_.emplace(piecewise_construct,
forward_as_tuple(rid), forward_as_tuple());
    }
  }else {
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),
AbortReason::kLockOnShrinking);
  }
}
```

C++11 中 std::piecewise_construct 的使用

1. 检查事务状态：

   首先，函数检查事务是否处于`kShrinking`状态，即事务是否正在缩小其影响范围。如果事务正在下降，那么不允许进行新的锁请求。

2. 事务未处于下降状态：

   如果事务不在缩小状态，函数继续检查`lock_table_`

   中是否已经存在对应`rid`的锁请求条目：如果不存在，使用`emplace`方法在`lock_table_`中创建一个新的条目。这里使用`piecewise_construct`和`forward_as_tuple`是为了使用完美转发，允许构造函数接受不同类型的参数。

3. 事务处于下降状态：

   如果事务处于`kShrinking`状态，函数将事务状态设置为`kAborted`，并抛出`TxnAbortException`异常。这表示在事务下降期间尝试进行锁请求是不允许的，并且请求将被中止。

4. 异常信息：

   异常包含了事务ID和中止原因`kLockOnShrinking`，有助于调用者理解事务为何被中止。

CheckAbort

`CheckAbort(Txn, LockRequestQueue)`：检查txn的state是否是abort，如果是，做出相应的操作

函数实现：

```
void LockManager::CheckAbort(Txn *txn,
LockManager::LockRequestQueue &req_queue) {
  //because LockX & LockS remove the other information after
'check'
  //so we don't correct the other information
  if(txn->GetState() == TxnState::kAborted){
    req_queue.EraseLockRequest(txn->GetTxnId());
    throw TxnAbortException(txn-
>GetTxnId(),AbortReason::kDeadlock);
  }
}
```

1. 检查事务状态：

   函数首先检查传入的事务（`txn`）的状态。如果事务状态是`kAborted`，表示事务已经被中止。

2. 处理中止的事务：

   如果事务已被中止，执行以下操作：

   - 调用`req_queue`的`EraseLockRequest`方法，传入事务ID，从锁请求队列中移除该事务的锁请求。这一步是为了清理资源，避免中止事务继续占用锁资源。

3. 抛出异常：

   抛出`TxnAbortException`异常，异常中包含了事务ID和中止原因`kDeadlock`。这里使用`kDeadlock`作为中止原因可能是出于简化处理的考虑，但实际中止原因可能因多种情况而异，例如违反隔离级别规则、锁等待超时等。

4. 在排他锁（`LockX`）和共享锁（`LockS`）操作后，某些信息会在检查之后被移除，因此在`CheckAbort`中不需要对这些信息进行修正。

AddEdge

函数实现：

```
void LockManager::AddEdge(txn_id_t t1, txn_id_t t2) {
  waits_for_[t1] .emplace(t2);
}
```

向等待图中插入边，利用`emplace`向`map`中插入元素。

RemoveEdge

函数实现：

```
void LockManager::RemoveEdge(txn_id_t t1, txn_id_t t2) {
  waits_for_[t1].erase(t2);
}
```

向等待图中插入边，利用`erase`向`map`中删除元素。

HasCycle

函数实现：

```
bool LockManager::HasCycle(txn_id_t &newest_tid_in_cycle) {
  visited_set_.clear();
  stack<txn_id_t> empty_stack;
  visited_path_.swap(empty_stack);
  revisited_node_ = INVALID_TXN_ID;
  std::set<txn_id_t> all_txn;
  for(auto& iter:waits_for_){
    all_txn.emplace(iter.first);
    for(auto waited_node:iter.second){
      all_txn.emplace(waited_node);
    }
  }
  for(auto& wait_item:all_txn){
    if(DFS_Gwait(wait_item)){
      newest_tid_in_cycle = revisited_node_;
      while (!visited_path_.empty() && revisited_node_ !=
visited_path_.top()) {
        newest_tid_in_cycle = std::max(newest_tid_in_cycle,
visited_path_.top());
        visited_path_.pop();
      }
```

```
      return true;
    }
  }
  return false;
}
```

1. 初始化和重置状态：

   清空`visited_set_`，这是存储已访问事务的集合。

   将`visited_path_`与一个空栈`empty_stack`交换，以重置访问路径栈。

   将`revisited_node_`设置为一个无效的事务ID（`INVALID_TXN_ID`），用于记录在死锁循环中重复访问到的节点。

2. 收集所有事务ID：

   创建一个集合`all_txn`，用于存储所有事务的ID。

   遍历`waits_for_`映射，将每个等待事务的发起者和被等待的事务ID添加到`all_txn`中。

3. 检测每个事务：

   遍历`all_txn`中的每个事务ID`wait_item`。

4. 深度优先搜索检测死锁：

   对每个事务ID调用`DFS_Gwait`函数进行深度优先搜索，以检测是否存在等待循环。

5. 死锁检测结果：

   如果`DFS_Gwait`返回`true`，表示存在死锁循环。

   将`newest_tid_in_cycle`设置为当前循环中检测到的最近事务ID（`revisited_node_`）。

6. 更新最近事务ID：

   从栈`visited_path_`中弹出元素，直到栈为空或栈顶元素等于`revisited_node_`。

   在弹出的过程中，使用`std::max`函数更新`newest_tid_in_cycle`，确保它始终是循环中最新出现的事务ID。

7. 返回死锁结果：

   如果在任何事务上检测到死锁，函数返回`true`。

   如果遍历完所有事务都没有检测到死锁，函数返回`false`。

DeleteNode

函数实现：

```
void LockManager::DeleteNode(txn_id_t txn_id) {
    waits_for_.erase(txn_id);

    auto *txn = txn_mgr_->GetTransaction(txn_id);

    for (const auto &row_id: txn->GetSharedLockSet()) {
        for (const auto &lock_req:
lock_table_[row_id].req_list_) {
            if (lock_req.granted_ == LockMode::kNone) {
                RemoveEdge(lock_req.txn_id_, txn_id);
            }
        }
    }
    for (const auto &row_id: txn->GetExclusiveLockSet()) {
        for (const auto &lock_req:
lock_table_[row_id].req_list_) {
            if (lock_req.granted_ == LockMode::kNone) {
                RemoveEdge(lock_req.txn_id_, txn_id);
            }
        }
    }
}
```

1. 删除等待关系：

   首先，函数从`waits_for_`映射中删除与`txn_id`相关的所有等待关系。这意味着如果其他事务正在等待这个事务释放锁，这些等待关系将不再存在。

2. 获取事务对象：

   通过事务管理器（`txn_mgr_`）获取与`txn_id`关联的事务对象。

3. 处理共享锁集合：

   遍历事务的共享锁集合`GetSharedLockSet`，对于集合中的每个行ID`row_id`：

   - 遍历该行ID在锁表（`lock_table_`）中的锁请求列表`req_list_`。
   - 对于每个锁请求，如果锁请求的授予状态是`LockMode::kNone`（表示没有授予锁），则调用`RemoveEdge`函数来删除从当前锁请求事务`lock_req.txn_id_`指向`txn_id`的依赖边。

4. 处理排他锁集合：

   类似于共享锁集合的处理，遍历事务的排他锁集合`GetExclusiveLockSet`，对于集合中的每个行ID：

   - 遍历该行ID在锁表中的锁请求列表。
   - 对于每个锁请求，如果锁请求的授予状态是`LockMode::kNone`，则删除指向`txn_id`的依赖边。

5. 删除事务的锁请求：

   通过上述步骤，函数删除了所有与`txn_id`相关的锁请求，这些请求要么因为事务结束而不再需要，要么因为事务没有获得锁而需要从依赖图中移除。

RunCycleDetection

函数实现：

```
void LockManager::RunCycleDetection() {
  while(enable_cycle_detection_==true) {
    this_thread::sleep_for(cycle_detection_interval_);
    unique_lock<mutex> lock_latch(latch_);
    waits_for_.clear();
    // construct the graph
    // find all txn
    unordered_map<RowId, set<txn_id_t>> data_txn_locked;
    unordered_map<RowId, set<txn_id_t>> data_txn_unlocked;
    for (auto& iter : lock_table_) {
      //data_id
      for (auto& request : iter.second.req_list_) {
        //txn_id
        //1.grantee locked
        if(request.granted_ != LockMode::kNone) {
          data_txn_locked[iter.first].emplace(request.txn_id_);
        }else{

 data_txn_unlocked[iter.first].emplace(request.txn_id_);
        }
      }
    }
    for (auto &iter : lock_table_) {
      //data_id
```

```
      for (auto& request : iter.second.req_list_) {
        //txn_id
        //2.addedge
        if(request.granted_ == LockMode::kNone) {
          for(auto grant:data_txn_locked[iter.first]) {
            AddEdge(request.txn_id_, grant);
          }
        }
      }
    }
    txn_id_t txn_id = INVALID_TXN_ID;
    //detect cycle
    while(HasCycle(txn_id)==true){
      DeleteNode(txn_id);
      txn_mgr_->GetTransaction(txn_id)->SetState(TxnState::kAborted);
      for(auto item:data_txn_unlocked){
        if(item.second.find(txn_id)!=item.second.end()){
          lock_table_[item.first].cv_.notify_all();
        }
      }
    }
  }
}
```

1. 循环检测死锁：

   只要`enable_cycle_detection_`标志为`true`，函数就会继续运行。

2. 休眠等待：

   使用`this_thread::sleep_for`函数根据`cycle_detection_interval_`指定的时间间隔休眠，以减少检测频率，避免过度占用资源。

3. 获取互斥锁：

   使用`unique_lock`获取`latch_`互斥锁，以确保在构建图和检测死锁时的线程安全。

4. 清空等待关系：

   清空`waits_for_`映射，为构建新的依赖图做准备。

5. 构建依赖图：

   创建两个哈希表`data_txn_locked`和`data_txn_unlocked`，分别存储每个数据项上已授予锁和未授予锁的事务集合。

遍历`lock_table_`中的每个锁请求：

- 如果事务已被授予锁（`granted_ != LockMode::kNone`），将其添加到`data_txn_locked`中。
- 如果事务未被授予锁（`granted_ == LockMode::kNone`），将其添加到`data_txn_unlocked`中。

6. 添加依赖边：

   再次遍历`lock_table_`中的每个锁请求：

   - 如果事务未被授予锁，对于数据项上已被授予锁的每个事务，添加一条从等待锁的事务到授予锁的事务的依赖边（`AddEdge`）。

7. 死锁检测：

   使用`HasCycle`函数进行死锁检测，如果发现死锁，记录死锁循环中的最近事务ID（`txn_id`）。

8. 处理死锁：

   如果检测到死锁，执行以下操作：

   - 调用`DeleteNode`函数删除死锁事务的所有锁请求和依赖边。
   - 设置死锁事务的状态为`kAborted`。
   - 遍历`data_txn_unlocked`，如果事务在未解锁事务集合中，通知所有等待该数据项的事务（`notify_all`）。

9. 循环处理死锁：

   继续检测死锁，直到没有发现死锁为止。

DFS_Gwait

函数实现：

```
bool LockManager::DFS_Gwait(txn_id_t txn_id){
   if(visited_set_.find(txn_id) != visited_set_.end()){
     for(auto& item : visited_set_){
       revisited_node_ = revisited_node_>item?
revisited_node_:item;
     }
     txn_id = revisited_node_;
     return true;
   }
   visited_set_.emplace(txn_id);
   visited_path_.push(txn_id);
    for(auto iter:waits_for_[txn_id]){
```

```
12          if(DFS_Gwait(iter)){
13            return true;
14          }
15        }
16        visited_path_.pop();
17        visited_set_.erase(txn_id);
18        return false;
19      }
```

1. **检查事务是否已访问：**

   函数首先检查当前事务ID（`txn_id`）是否已经在`visited_set_`中。这用于避免无限递归和循环。

2. **处理已访问事务：**

   如果事务已访问，函数遍历`visited_set_`中的所有事务，并使用一个变量`revisited_node_`来存储已经访问过的的事务ID。然后，将`txn_id`设置为这个事务ID，并返回`true`，表示存在循环等待。

3. **标记事务为已访问：**

   如果事务未访问，将其添加到`visited_set_`中。

4. **添加事务到访问路径：**

   将当前事务ID添加到`visited_path_`栈中，用于记录访问路径。

5. **递归检查等待事务：**

   遍历当前事务等待的事务列表（`waits_for_[txn_id]`），对每个等待的事务递归调用`DFS_Gwait`函数。

6. **检查死锁：**

   如果在递归调用中发现死锁（即返回`true`），则当前函数也返回`true`。

7. **回溯并清理状态：**

   在完成当前事务的所有递归检查后，从`visited_path_`栈中弹出当前事务ID，并从`visited_set_`中移除该事务ID，以进行回溯。

遇到的问题及解决方法

在本节中遇到的问题主要是对头文件的阅读和条件变量的使用，等待图的遍历DFS就可以解决，难度不大。这里主要说明下对于LM的设计想法：

- 先准备查询事务本身的状态，如果不符合上锁解锁的条件要抛出异常
- 确定可以修改之后，按照修改事务本身状态->LM状态->等待图构建所需变量

&写报告写太累了，就不展开细说我遇到的很多大坑了（

思考题

> *本模块中，为了简化实验难度，我们将Lock Manager模块独立出来。如果不独立出来，做到并发查询期间根据指定的隔离级别进行事务的边界控制，考虑模块3中B+树并发修改的情况，需要怎么设计？*
>
> *注：如果完成了本模块，请在实验报告里完成思考题。思考题占本模块30%的分数，请尽量回答的详细些，比如具体到涉及哪些模块、哪些函数的改动，大致怎样改动。有能力、有时间的同学也可以挑战一下直接在代码上更改。*

要控制好不同隔离等级下的事务边界控制，最重要是掌握好对数据的上锁解锁，我们可以借助教材提到的意向锁的机制来实现对数据的访问权限授予，意向锁的作用可以不仅限与页表等多级数据，也可以完全适用到B+树

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

当我们在利用B+索引进行插入、删除、查询的时候，我们可以利用意向锁的机制查看当前数据是否空闲，来实现不同隔离等级下的对数据的访问

具体操作函数位置放在索引查找的函数的地方，新建变量标识现在的意向锁，再通过条件变量控制当前线程的并发控制，当此时的意向锁解绑或是降级之后，通过`notify`方法通知并行进程。

总结

不知不觉已经写到这里了，每一次打开Clion的时候都不知道自己能写到哪里，好在我有可靠的小组成员和我一起Debug、有悉心指导的聂俊哲&石宇新助教，在数据库学习中感谢孙建伶老师的悉心教导，模块7就写到这里，是一次酣畅淋漓的C++工程管理和数据库学习。

今天是6月8日高考第二日，祝全国考生以及母校平遥中学的学弟学妹们高考顺利，前程似锦！

愿旅途臻若星河，灿如骄阳！