7 LOCK MANAGER

7.1 实验概述

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。

Bonus：本模块涉及锁和条件变量，独立开发难度较大，属于Bonus。因本模块独立于验收流程外，完成的验收时会有专门针对本模块的提问环节考察理解。

7.2 事务管理器

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。

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`的代码已经为大家实现好了，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。

7.3 锁管理器

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`.

1. `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`。

1. `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。

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。

你需要修改的是`LockManager`类(`concurrency/lock_manager.cpp,concurrency/lock_manager.h`)中以下几个函数：

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

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。

7.4 死锁检测

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。

7.5 模块相关代码

- `src/include/concurrency/txn.h`
- `src/include/concurrency/txn_manager.h`
- `src/include/concurrency/lock_manager.h`
- `src/concurrency/lock_manager.cpp`
- `src/concurrency/txn_manager.cpp`

- `test/concurrency/lock_manager_test.cpp`

7.6 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。

7.7 诚信守则

1. 请勿从其它组或在网络上找到的其它来源中复制源代码，一经发现抄袭，成绩为0；
2. 请勿将代码发布到公共Github存储库上。