浙江大學

数据库系统实验报告

题　　目：	MiniSQL
授课教师：	孙建伶
助　　教：	聂俊哲/石宇新
姓　　名：	王淳
学　　号：	3220105023
邮　　箱：	zjuheadmaster@zju.edu.cn
联系电话：	13428807817

Contents

1 MiniSQL系统概述

1.1 前言

如果将数据库比作一座大厦，那么学习SQL语言就像学习如何使用这座大厦的各种设施。我们能够执行一些基本的SQL操作，就如同我们能够在大厦内轻松找到电梯、使用会议室、或进入办公室。然而，我们对数据库系统的理解仍然很表面，就像我们对大厦的建筑结构和基础设施知之甚少。

而编写miniSQL的过程则类似于设计和建造一座大厦。从这个过程中，我们可以深入了解数据库系统（DBMS）的运行原理。从最基本的内存管理、记录处理开始，逐步涉及到索引的创建与搜索，再到执行计划的生成与选择，最后到实际执行。通过编写miniSQL，我们不仅是学习如何实现一个简单的数据库，更是深入理解和巩固数据库理论知识。

这不仅大大提升了我们的实际操作能力，还加深了我们对数据库系统内在机制的理解。就像亲自设计和建造一座大厦能让我们全面理解其每一个结构和功能部件的协同工作原理，编写miniSQL也能使我们更透彻地理解数据库系统的各个层面。这对我们的实践能力提升和理论知识深化都有极大的帮助。

1.2 功能描述

1. 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
2. 表定义：一个表可以定义多达32个属性，各属性可以指定是否为`unique`，支持单属性的主键定义。
3. 索引定义：对于表的主属性自动建立B+树索引，对于声明为`unique`的属性也需要建立B+树索引。
4. 数据操作: 可以通过`and`或`or`连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。
5. 良好的性能：我们实现了一个工业级别的B+树，可以毫无压力的处理10w+的数据。

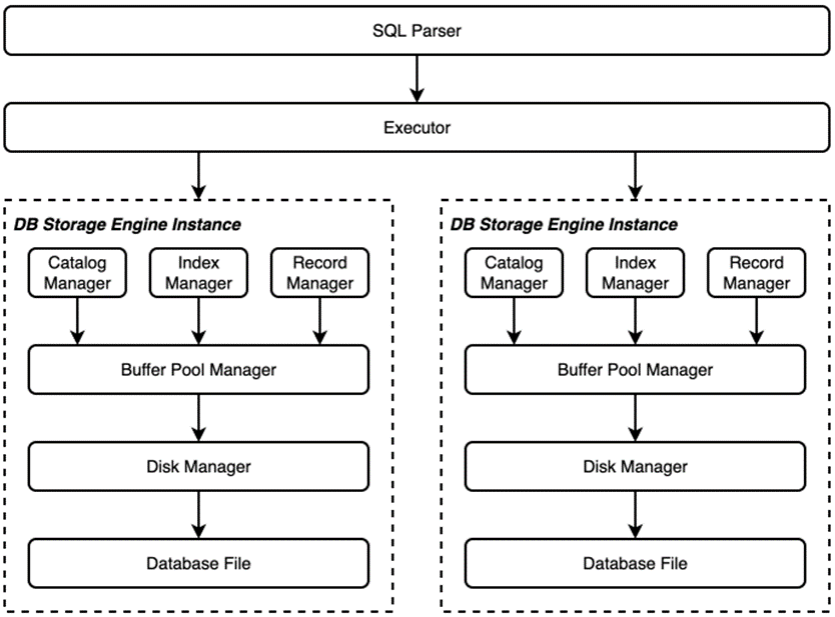
1.3 运行环境

使用Clion连接Win11下的WSL或者Linux服务器。

1.4 参考资料

在ZJU-Git上的框架以及15445课程理论

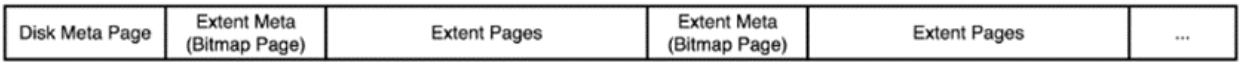
2 MiniSQL系统结构设计



如上图所示的系统架构，解释器SQL Parser在解析SQL语句后将生成的语法树交由执行器Executor处理。执行器则根据语法树的内容对相应的数据库实例（DB Storage Engine Instance）进行操作，而每个DB Storage Engine Instance对应了一个数据库实例（即通过CREATE DATABSAE创建的数据库）。在每个数据库实例中，用户可以定义若干表和索引，表和索引的信息通过Catalog Manager、Index Manager和Record Manager进行维护。

2.1 Disk Manager模块

在MiniSQL的设计中，Disk Manager和Buffer Pool Manager模块位于架构的最底层。Disk Manager主要负责数据库文件中数据页的分配和回收，以及数据页中数据的读取和写入。其中，数据页的分配和回收通过位图（Bitmap）这一数据结构实现，**位图中每个比特（Bit）对应一个数据页的分配情况**，用于标记该数据页是否空闲（`0`表示空闲，`1`表示已分配）。当Buffer Pool Manager需要向Disk Manager请求某个数据页时，Disk Manager会通过某种映射关系，找到该数据页在磁盘文件中的物理位置，将其读取到内存中返还给Buffer Pool Manager。而Buffer Pool Manager主要负责将磁盘中的数据页从内存中来回移动到磁盘，这使得我们设计的数据库管理系统能够支持那些占用空间超过设备允许最大内存空间的数据库。



2.2 Buffer Pool Manager模块

其他模块函数并不直接与Disk产生交互，都需要通过Buffer Pool（内存池）来实现具体Disk数据读取，Buffer Pool Manager根据需求将硬盘中的数据分配到内存中，供调用者使用。**Buffer Pool Manager中的操作对数据库系统中其他模块是不透明的。**例如，在系统的其它模块中，可以使用数据页唯一标识符`page_id`向Buffer Pool Manager请求对应的数据页。但实际上，这些模块并不知道该数据页是否已经在内存中还是需要从磁盘中读取。同样地，Disk Manager中的数据页读写操作对Buffer Pool Manager模块也是透明的，即Buffer Pool Manager使用逻辑页号`logical_page_id`向Disk Manager发起数据页的读写请求，但Buffer Pool Manager并不知道读取的数据页实际上位于磁盘文件中的哪个物理页（对应页号`physical_page_id`）。

数据库系统中，所有内存页面都由`Page`对象表示，每个`Page`对象都包含了一段连续的内存空间`data_`和与该页相关的信息（如是否是脏页，页的引用计数等等）。注意，`Page`对象并不作用于唯一的数据页，它只是一个用于存放从磁盘中读取的数据页的容器。这也就意味着同一个`Page`对象在系统的整个生命周期内，可能会对应很多不同的物理页。`Page`对象的唯一标识符`page_id_`用于跟踪它所包含的物理页，如果`Page`对象不包含物理页，那么`page_id_`必须被设置为`INVALID_PAGE_ID`。每个`Page`对象还维护了一个计数器`pin_count_`，它用于记录固定(Pin)该页面的线程数。Buffer Pool Manager将不允许释放已经被固定的`Page`。每个`Page`对象还将记录它是否脏页，在复用`Page`对象之前必须将脏的内容转储到磁盘中。

其主要实现函数有`FetchPage`、`NewPage`、`UnpinPage`等；在这里我们需要指出，内存的管理策略，是先从内存池中寻找数据页是否已被读入，若失败再从free_list中寻找可以使用的页，最后再考虑使用替换策略。

另外，需要特别注意的是，我们只关注于dirty数据页的状况，因为只有dirty的数据页才会影响disk的数据存储情况；且在`unpin`函数中，我们需要维护一`pin_count`变量；因为可能存在多个进程调用统一数据页的情况，所以当`pin_count`数值不为0时，我们只是减少了其数值大小，并没有真正的释放数据页。

在本模块中，我们最先使用的替换策略是`lru_replacer`。`Least Recently Used`算法，将最近最少使用的数据页回收。最开始的实现是使用了一个双向链表进行记录。

链表容量为内存能容纳的最大数据页数。将内存中已经不再引用的数据页号放入链表，当内存满时，可以回收一个页来开辟出新空间时，把队列最前端（最早放置）的页号对应的页回收。需要注意，当复用链表中的页时，需要将其从队列中取出，并重新放入链表的最后端（表示最近使用）。

但在具体的实现过程中我们发现，因为lru_placer的遍历策略，会使得Pin函数大量占用插入语句的运行带宽，造成极大的时间消耗；因而在此基础之上实现了Bonus：clock replacer，它的实现相对简单，使用一个循环缓冲区和一个指针来跟踪页面。每次需要替换页面时，只需检查指针指向的页面的使用位，如果使用位为 0 则替换该页面，否则将使用位置为 0 并移动指针到下一个页面。这种实现方式使得 Clock Replacer 的时间复杂度接近 $O(1)$。

2.3 Record Manager模块

在minisql的设计中，Record Manager负责管理数据表中所有的记录（精确到行），它能够支持记录的插入、删除与查找操作，并对外提供相应的接口。

对于record记录而言，我们存有以下几个相关概念（对映于数据库中表的构成）：

1. 列，用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等；
2. 模式，用于表示一个数据表或是一个索引的结构，其中一个模式由一个或多个列组成；
3. 域，对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；
4. 行，与元组的概念等价，用于存储记录或索引键，其中一行由一个或多个域组成。

在minisql的数据表存储中，我们为了方便统一处理，都需要提供对映的函数将上面提到的行，域，列，模式四个类型转化为字节流对象，以写入数据页中。另外，为了能够从磁盘中恢复这些对象，我们同样需要能够提供一种反序列化的方法，把字节流对象转换为我们需要的对象类型。

序列化（Serialization）	是将数据结构或对象转换成一种可存储或可传输格式的过程。在序列化后，数据可以被写入文件、发送到网络或存储在数据库中，以便在需要时可以再次还原成原始的数据结构或对象。序列化的过程通常涉及将数据转换成字节流或类似的格式，使其能够在不同平台和编程语言之间进行传输和交换。
反序列化（Deserialization）	是序列化的逆过程，即将序列化后的数据重新还原成原始的数据结构或对象。反序列化是从文件、网络数据或数据库中读取序列化的数据，并将其转换回原始形式，以便在程序中进行使用和操作。

另外此模块还涉及了Table Heap这一重要概念。一个table heap里面用双向链表的储存了大量的table page，而table page里又存储了一个又一个row（tuple/entity）。在此我们可以使用一个Rowid定位到一个唯一的Row，通过高32位获取对映的page_id，低32位获取Row在该page中的对映位置（slot_num），从而介导执行器的实际操作。

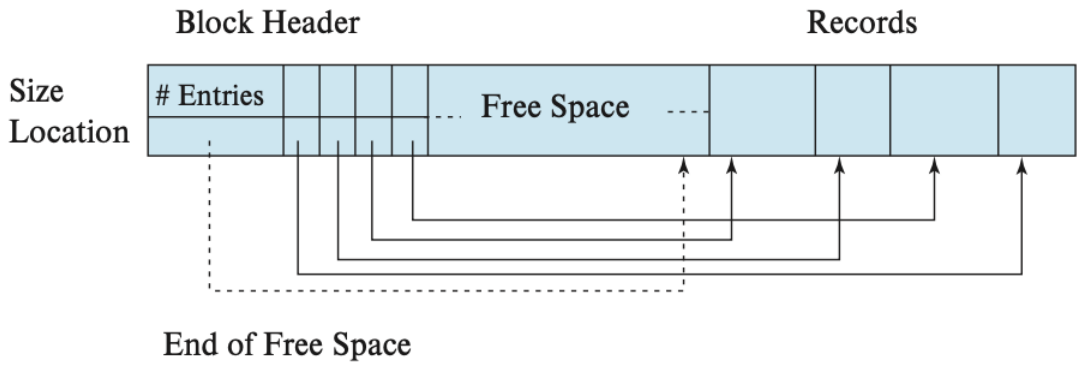


Figure 13.6 Slotted-page structure.

在这部分，我们主要实现了InsertTuple、UpdateTuple、ApplyDelete等函数，以及TableIterator迭代器，供于执行器进行简便地调用与实现。

2.4 Index Manager模块

Index Manager 负责数据表索引的实现和管理，包括：索引的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。

容易注意到，通过遍历堆表的方式来查找一条记录是十分低效的。为实现数据的快速定位，基于磁盘的B+树动态索引结构是一个好的选择，可以支持随机查找和高效访问有序记录。此模块负责数据表索引的实现和管理，包括：索引的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。

B+树中的每个结点（Node）都对应一个数据页，用于存储B+树结点中的数据。因此在本节中，你需要实现以下三种类型的B+树结点数据页：

2.4.1 BPlusTreePage

`BPlusTreePage`是`BPlusTreeInternalPage`和`BPlusTreeLeafPage`类的公共父类，它包含了中间结点和叶子结点共同需要的数据：

- `page_type_`：标记数据页是中间结点还是叶子结点；
- `key_size_`：当前索引键的长度；
- `lsn_`：数据页的日志序列号，该模块中不会用到；
- `size_`：当前结点中存储Key-Value键值对的数量；
- `max_size_`：当前结点最多能够容纳Key-Value键值对的数量；
- `parent_page_id_`：父结点对应数据页的`page_id`；
- `page_id_`：当前结点对应数据页的`page_id`。

你需要在`src/include/page/b_plus_tree_page.h`和`src/page/b_plus_tree_page.cpp`中实现`BPlusTreePage`类。

2.4.2 BPlusTreeInternalPage

中间结点`BPlusTreeInternalPage`不存储实际的数据，它只按照顺序存储m个键和$m+1$个指针（这些指针记录的是子结点的`page_id`）。由于键和指针的数量不相等，因此我们需要将第一个键设置为INVALID，也就是说，顺序查找时需要从第二个键开始查找。在任何时候，每个中间结点至少是半满的（Half Full）。当删除操作导致某个结点不满足半满的条件，需要通过合并（Merge）相邻两个结点或是从另一个结点中借用（移动）一个元素到该结点中（Redistribute)来使该结点满足半满的条件。当插入操作导致某个结点溢出时，需要将这个结点分裂成为两个结点。

为了便于理解和设计，我们将键和指针以`pair`的形式顺序存储，但由于键和指针的数量不一致，我们不得已牺牲一个键的空间，将其标记为INVALID。也就是说对于B+树的每一个中间结点，我们都付出了一个键的空间代价。实际上有一种更为精细的设计选择：定义一个大小为m的数组连续存放键，然后定义一个大小为$m+1$的数组连续存放指针，这样设计的好处在于，一是没有空间上的浪费，二是在键值查找时CPU缓存的命中率较高（局部性原理)。

2.4.3 PlusTreeLeafPage

叶结点`BPlusTreeLeafPage`存储实际的数据，它按照顺序存储m个键和m个值，其中键由一个或多个`Field`序列化得到，在`BPlusTreeLeafPage`类中用模板参数`KeyType`表示；值实际上存储的是`RowId`的值，它在`BPlusTreeLeafPage`类中用模板参数`ValueType`表示。叶结点和中间结点一样遵循着键值对数量的约束，同样也需要完成对应的合并、借用和分裂操作。

此外，需要补充一些基本概念，索引键(Key)是索引列的值序列化后得到的字符串,值(Value)根据结点的类型而有所不同，其中叶节点储存row_id信息，而内部节点存储的是page_id。KeyManager负责对GenericKey进行序列化/反序列化比较。

B+树索引目前只支持unique_key，在出现插入的key-value键值对出现重复时，会导致索引更新失败。B+树目前支持创建、插入、删除、查找和释放等操作，其遍历通过迭代器实现，与堆表实现类似，B+树索引迭代器可以在范围查询时将所有的叶结点组织成为一个单向链表，然后沿着特定方向有序遍历叶结点数据页中的每个键值对。

2.5 Catalog Manager模块

在内存中，以TableInfo和IndexInfo的形式存储表和索引，从而维护与之对应的表和索引的元信息和操作对象。对于索引来说，其元信息是在定义时被存储的，而其表信息、模式信息都是在反序列化（具体序列化和反序列化的解释见Record Manager模块介绍）后元信息生成的。因此为了便于维护所有表和索引的定义信息的durability（持久化性）和recoverability（可恢复性:在重启时从数据库文件中恢复），我们在元信息类TableMetadata和IndexMetadata实现序列化和反序列化操作。

为每一个表和索引分配一个单独数据页是一个较为简单的处理方式，因此我们同样需要一个数据页和数据对象CatalogMeta来记录管理元信息存储与数据页的对应关系。本节主要是在既定的序列化和反序列化方法中确定序列化所需的存储大小，包括三个类CatalogMeta、IndexMetadata、TableMetadata，需要它们对应的GetSerializedSize()的返回值。

CatalogManager类具备维护和持久化数据库中所有表和索引的信息，能够在数据库实例（DBStorageEngine）初次创建时初始化元数据，并在后续重新打开数据库实例时，从数据库文件中加载所有的表和索引信息，构建TableInfo和IndexInfo信息置于内存中。

- 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
- 表中每个字段的定义信息，包括字段类型、是否唯一等。
- 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

这些模式信息在被创建、修改和删除后还应被持久化到数据库文件中。此外，Catalog Manager还需要为上层的执行器Executor提供公共接口以供执行器获取目录信息并生成执行计划。

此外，CatalogManager类部分接口，如CreateTable、GetTable、GetTables、DropTable、GetTableIndexes可以对上层模块（Executor）提供对数据表的操作方式，而CreateIndex、GetIndex、DropIndex则是可以对上层模块提供对指定索引的操作方式。

2.6 Exeuctor模块

在拿到 Planner 生成的具体的查询计划后，就可以生成真正执行查询计划的一系列算子了。生成算子的步骤很简单，遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor。算子的执行模型也大致分为三种：

1. Iterator Model，即经典的火山模型。执行引擎会将整个 SQL 构建成一个 Operator 树，查询树自顶向下的调用接口，数据则自底向上的被拉取处理。每一种操作会抽象为一个 Operator，每个算子都有 Init() 和 Next() 两个方法。Init() 对算子进行初始化工作。Next() 则是向下层算子请求下一条数据。当 Next() 返回 false 时，则代表下层算子已经没有剩余数据，迭代结束。

    (a) 该方法的优点是其计算模型简单直接，通过把不同物理算子抽象成一个个迭代器。每一个算子只关心自己内部的逻辑即可，让各个算子之间的耦合性降低，从而比较容易写出一个逻辑正确的执行引擎。

    (b) 缺点是火山模型一次调用请求一条数据，占用内存较小，但函数调用开销大，特别是虚函数调用造成 cache miss 等问题。同时，逐行地获取数据会带来过多的 I/O，对缓存也不友好。

2. Materialization Model，算子计算出所有结果后一起返回。这种模型的弊端显而易见，当数据量较大时，内存占用很高。但该模型减少了函数调用的开销。比较适合查询数据量较小的 OLTP workloads。

3. Vectorization Model. 对上面两种模型的中和，输入和输出都以Batch为单位。在Batch的处理模式下，计算过程还可以使用SIMD指令进行加速。目前比较先进的 OLAP 数据库都采用这种模型。

本任务采用的是最经典的 Iterator Model。在本次任务中，我们实现了5个算子，分别是select，Index Select，insert，update，delete。 对于每个算子，都实现了 Init 和 Next 方法。 Init 方法初始化运算符的内部状态，Next 方法提供迭代器接口，并在每次调用时返回一个元组和相应的 RID。对于每个算子，我们假设它在单线程上下文中运行，并不需要考虑多线程的情况。每个算子都可以通过访问 `ExecuteContext` 来实现表的修改，例如插入、更新和删除。 为了使表索引与底层表保持一致，插入删除时还需要更新索引。

2.7 Recovery Manager模块

Recovery Manager负责管理和维护数据恢复的过程，包括：

- 日志结构的定义
- 检查点CheckPoint的定义
- 执行Redo、Undo等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态

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，同组成员的工作之间影响过深，我们将Recovery Manager模块单独拆了出来。

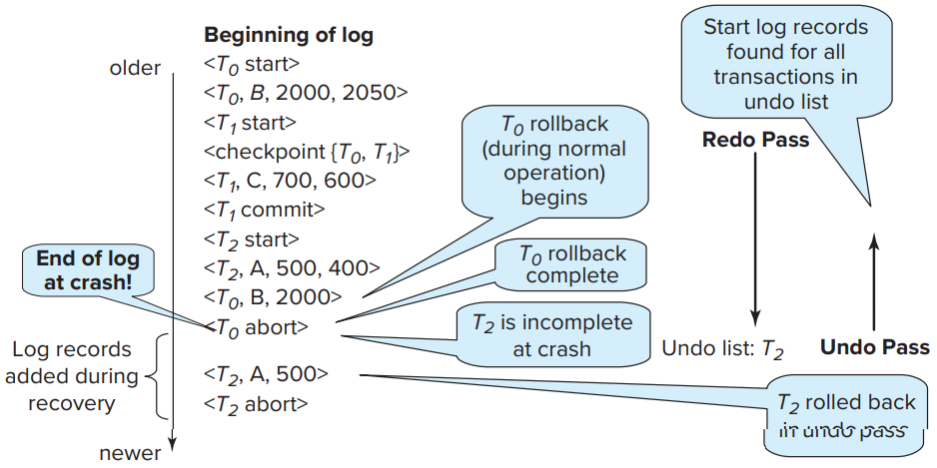
数据恢复是一个很复杂的过程，需要涉及系统的多个模块。以InnoDB为例，在其恢复过程中需要redo log、binlog、undo log等参与，这里把InnoDB的恢复过程主要划分为两个阶段：第一阶段主要依赖于redo log的恢复，而第二阶段需要binlog和undo log的共同参与。

第一阶段，数据库启动后，InnoDB会通过redo log找到最近一次checkpoint的位置，然后根据checkpoint相对应的LSN开始，获取需要重做的日志，接着解析日志并且保存到一个哈希表中，最后通过遍历哈希表中的redo log信息，读取相关页进行恢复。

在该阶段中，所有被记录到redo log但是没有完成数据刷盘的记录都被重新落盘。然而，InnoDB单靠redo log的恢复是不够的，因为数据库在任何时候都可能发生宕机，需要保证重启数据库时都能恢复到一致性的状态。这个一致性的状态是指此时所有事务要么处于提交，要么处于未开始的状态，不应该有事务处于执行了一半的状态。所以我们可以通过undo log在数据库重启时把正在提交的事务完成提交，活跃的事务回滚，保证了事务的原子性。此外，只有redo log还不能解决主从数据不一致等问题。

第二阶段，根据undo中的信息构造所有未提交事务链表，最后通过上面两部分协调判断事务是否需要提交还是回滚。InnoDB使用了多版本并发控制(MVCC)以满足事务的隔离性，简单的说就是不同活跃事务的数据互相是不可见的，否则一个事务将会看到另一个事务正在修改的数据。InnoDB借助undo log记录的历史版本数据，来恢复出对于一个事务可见的数据，满足其读取数据的请求。

在我们的实验中，日志在内存中以`LogRec`的形式表现，出于实现复杂度的考虑，我们将Recovery Manager模块独立出来，不考虑日志的落盘，用一个unordered_map简易的模拟一个KV Database，并直接在内存中定义一个能够用于插入、删除、更新，事务的开始、提交、回滚的日志结构。`CheckPoint`检查点应包含当前数据库一个完整的状态，`RecoveryManager`则包含`UndoPhase`和`RedoPhase`两个函数，代表Redo和Undo两个阶段。



2.8 Lock Manager模块

通过实现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。

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，实现的LM需要支持三种不同的隔离级别。

在`LockManager`类中以下几个函数：

- `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

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。

3 验收与检验流程

PASSED IS ALL YOU NEED

```
[----------] Global test environment tear-down
[==========] 31 tests from 12 test suites ran. (77889 ms total)
[  PASSED  ] 31 tests.

Process finished with exit code 0
```

1. 创建数据库`db0`、`db1`、`db2`，并列出所有的数据库

```
wsl: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost
StarSQL>show databases;
 show databases;
+----------+
| Database |
+----------+
| db2      |
| db1      |
| db0      |
+----------+
Current time: Mon Jun 10 23:27:38 2024
```

```
StarSQL>show databases;
 show databases;
+----------+
| Database |
+----------+
| db2      |
| db1      |
| db0      |
+----------+
Current time: Mon Jun 10 23:28:15 2024
StarSQL>drop database db0;
 drop database db0;
Current time: Mon Jun 10 23:28:41 2024
StarSQL>drop database db1;
 drop database db1;
Current time: Mon Jun 10 23:28:49 2024
StarSQL>drop database db2\;
 drop database db2\;
Minisql parse error at line 1, col 17, message: Unrecognized token [\] in input sql.
Unrecognized token [\] in input sql.
Current time: Mon Jun 10 23:28:52 2024
StarSQL>drop database db2;
 drop database db2;
Database not exists.
Current time: Mon Jun 10 23:28:57 2024
StarSQL>show databases;
 show databases;
Empty set (0.00 sec)
Current time: Mon Jun 10 23:29:02 2024
```

- drop掉重新建，建立`db0`和`db1`；

```
StarSQL>show databases;
 show databases;
Empty set (0.00 sec)
Current time: Mon Jun 10 23:29:02 2024
StarSQL>create database db0;
 create database db0;
Current time: Mon Jun 10 23:29:45 2024
StarSQL>create database db1;
 create database db1;
Current time: Mon Jun 10 23:29:54 2024
StarSQL>show databases;
 show databases;
+----------+
| Database |
+----------+
| db1      |
| db0      |
+----------+
```

2. 在`db0`数据库上创建数据表`account`，表的定义如下：

```
create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
```

```
StarSQL>use db0;
 use db0;
Database changed
Current time: Mon Jun 10 23:30:29 2024
StarSQL>create table account( id int, name char(16) unique, balance float, primary key(id));
 create table account( id int, name char(16) unique, balance float, primary key(id));
Current time: Mon Jun 10 23:30:39 2024
StarSQL>show tables;
 show tables;
+---------------+
| Tables_in_db0 |
+---------------+
| account       |
+---------------+
Current time: Mon Jun 10 23:30:46 2024
```

3. 考察SQL执行以及数据插入操作

执行数据库文件`sql.txt`，向表中插入100000条记录，批量执行时，所有sql执行完显示总的执行时间

```
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0020 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0020 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Current time: Mon Jun 10 23:37:22 2024
Query OK, 1 row affected(0.0010 sec).
Total time of file execution : 44.6650 sec.
Current time: Mon Jun 10 23:37:22 2024
```

4. 执行全表扫描`select * from account`，验证插入的数据是否正确（要求输出查询到100000条记录)

```
| 12599992 | name99992 | 598.010010 |
| 12599993 | name99993 | 587.750000 |
| 12599994 | name99994 | 314.859985 |
| 12599995 | name99995 | 970.500000 |
| 12599996 | name99996 | 243.809998 |
| 12599997 | name99997 | 477.829987 |
| 12599998 | name99998 | 449.160004 |
| 12599999 | name99999 | 303.079987 |
+----------+-----------+------------+
100000 row in set(0.5550 sec).
Current time: Mon Jun 10 23:45:30 2024
```

5. 考察点查询操作：

```sql
select * from account where id = 12599995;
select * from account where name = "name56789";
select * from account where id <> 12599995;
select * from account where balance <> 576.140015;
select * from account where name <> "name56769";
```

```
StarSQL>select * from account where id = 12599995;
 select * from account where id = 12599995;
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12599995 | name99995 | 970.500000 |
+----------+-----------+------------+
1 row in set(0.1780 sec).
Current time: Mon Jun 10 23:48:38 2024
StarSQL>select * from account where name = "name56789";
 select * from account where name = "name56789";
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12556789 | name56789 | 171.110001 |
+----------+-----------+------------+
1 row in set(0.1670 sec).
Current time: Mon Jun 10 23:48:50 2024
```

```
| 12599993 | name99993 | 587.750000 |
| 12599994 | name99994 | 314.859985 |
| 12599996 | name99996 | 243.809998 |
| 12599997 | name99997 | 477.829987 |
| 12599998 | name99998 | 449.160004 |
| 12599999 | name99999 | 303.079987 |
+----------+-----------+------------+
99999 row in set(0.5080 sec).
Current time: Mon Jun 10 23:49:44 2024
```

6. 考察多条件查询与投影操作：

```sql
select id, name from account where balance >= 185 and balance < 190;
select name, balance from account where balance > 125 and id <= 12599908;
select * from account where id < 12515000 and name > "name14500";
select * from account where id < 12500200 and name < "name00100";
```

1

```
| 12599472 | name99472 |
| 12599543 | name99543 |
| 12599596 | name99596 |
| 12599774 | name99774 |
| 12599988 | name99988 |
+----------+-----------+
500 row in set(0.1920 sec).
Current time: Mon Jun 10 23:54:04 2024
```

2

```
| name99900 | 301.589996 |
| name99901 | 466.839996 |
| name99903 | 826.380005 |
| name99905 | 307.619995 |
| name99906 | 964.710022 |
| name99907 | 800.219971 |
| name99908 | 229.860001 |
+-----------+------------+
87544 row in set(0.4140 sec).
Current time: Mon Jun 10 23:55:23 2024
```

3

```
| 12514996 | name14996 | 879.840027 |
| 12514997 | name14997 | 939.380005 |
| 12514998 | name14998 | 865.520020 |
| 12514999 | name14999 | 820.070007 |
+----------+-----------+------------+
499 row in set(0.1900 sec).
Current time: Mon Jun 10 23:55:59 2024
```

1

```
| 12599472 | name99472 |
| 12599543 | name99543 |
| 12599596 | name99596 |
| 12599774 | name99774 |
| 12599988 | name99988 |
+----------+-----------+
500 row in set(0.1920 sec).
Current time: Mon Jun 10 23:54:04 2024
```

4

```
| 12500097 | name00097 | 938.270020 |
| 12500098 | name00098 | 732.429993 |
| 12500099 | name00099 | 11.040000  |
+----------+-----------+------------+
100 row in set(0.2360 sec).
Current time: Mon Jun 10 23:56:19 2024
```

7. 考察唯一约束

```
create index idx01 on account(name);
select * from account where name = "name56789";#此处记录执行时间t2，要求t2<t1
select * from account where name = "name45678";#此处记录执行时间t3
select * from account where id < 12500200 and name < "name00100";
#此处记录执行时间t6，比较t5和t6
delete from account where name = "name45678";
insert into account values(?, "name45678", ?);
drop index idx01;          #执行(c)的语句，此处记录执行时间t4，要求 t3<t4
```

此处录制了视频（当时验收发生了小插曲），已经钉钉发送，打扰了助教哥哥，万分抱歉

8. 考察更新操作：`update account set id = ?, balance = ? where name = "name56789";` 并通过 `select` 操作验证记录被更新

```
StarSQL>update account set id = 12600001, balance = 10.23 where name = "name00001";
 update account set id = 12600001, balance = 10.23 where name = "name00001";
Query OK, 0 row affected(0.0000 sec).
Current time: Sun Jun 16 20:40:40 2024
StarSQL>select * from account where name = "name00001";
 select * from account where name = "name00001";
+----------+-----------+-----------+
| id       | name      | balance   |
+----------+-----------+-----------+
| 12600001 | name00001 | 10.230000 |
+----------+-----------+-----------+
1 row in set(0.0150 sec).
Current time: Sun Jun 16 20:41:18 2024
```

9. 考察删除操作：

(a) `delete from account where balance = ?`，并通过`select`操作验证记录被删除

(b) `delete from account`，并通过`select`操作验证全表被删除

(c) `drop table account`，并通过`show tables`验证该表

```
100 row in set(0.2210 sec).
BigTang minisql>delete from account where name = "name45678";
 delete from account where name = "name45678";
Query OK, 1 row affected(0.1900 sec).
BigTang minisql>insert into account values(228, "name45678", 401);
 insert into account values(228, "name45678", 401);
Query OK, 1 row affected(0.0030 sec).
BigTang minisql>
```

```
| 12599999 | name99999 | 303.079987 |
+----------+-----------+------------+
5 row in set(0.1940 sec).
BigTang minisql>delete from account;
 delete from account;
Query OK, 99998 row affected(3.7470 sec).
BigTang minisql>select *  from account;
 select *  from account;
Empty set(0.0040 sec).
BigTang minisql>
```

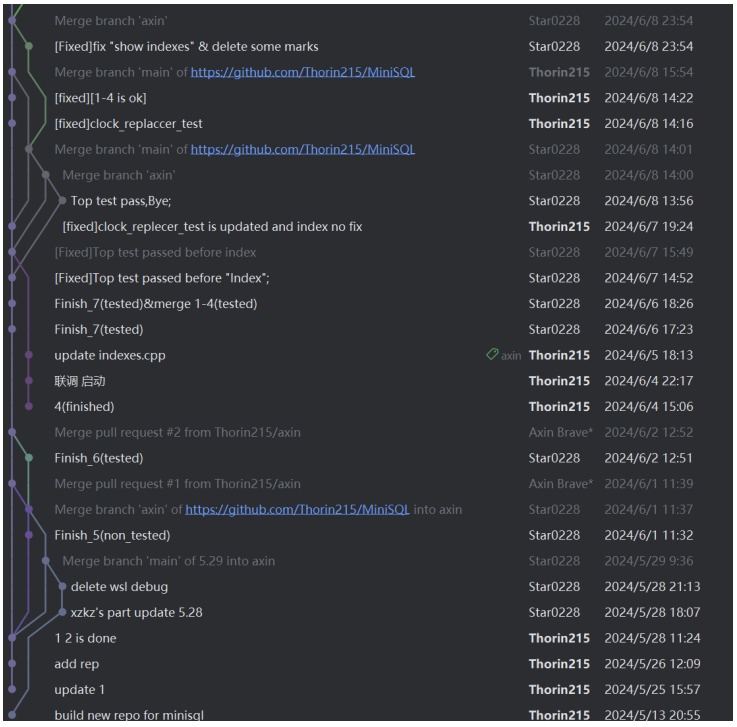
```
 select *  from account;
Empty set(0.0040 sec).
BigTang minisql>drop table account;
 drop table account;
BigTang minisql>show tables;
 show tables;
Empty set (0.00 sec)
BigTang minisql>
```

4 分组与设计分工

姓名	学号	分工
王淳	3220105023	1 4模块以及对应bonus以及小组报告
王晓宇	3220104364	5 6 7模块以及对应bonus
徐詹康哲	3220105799	2 3模块以及对应bonus

5 提交附录

- MiniSQL源代码
- 良好的Git记录
- 个人报告以及小组报告


Merge branch 'axin' Star0228 2024/6/8 23:54
[Fixed]fix "show indexes" & delete some marks Star0228 2024/6/8 23:54
Merge branch 'main' of https://github.com/Thorin215/MiniSQL Thorin215 2024/6/8 15:54
[fixed][1-4 is ok] Thorin215 2024/6/8 14:22
[fixed]clock_replaccer_test Thorin215 2024/6/8 14:16
Merge branch 'main' of https://github.com/Thorin215/MiniSQL Star0228 2024/6/8 14:01
Merge branch 'axin' Star0228 2024/6/8 14:00
Top test pass,Bye; Star0228 2024/6/8 13:56
[fixed]clock_replecer_test is updated and index no fix Thorin215 2024/6/7 19:24
[Fixed]Top test passed before index Star0228 2024/6/7 15:49
[Fixed]Top test passed before "Index"; Star0228 2024/6/7 14:52
Finish_7(tested)&merge 1-4(tested) Star0228 2024/6/6 18:26
Finish_7(tested) Star0228 2024/6/6 17:23
update indexes.cpp axin Thorin215 2024/6/5 18:13
联调 启动 Thorin215 2024/6/4 22:17
4(finished) Thorin215 2024/6/4 15:06
Merge pull request #2 from Thorin215/axin Axin Brave* 2024/6/2 12:52
Finish_6(tested) Star0228 2024/6/2 12:51
Merge pull request #1 from Thorin215/axin Axin Brave* 2024/6/1 11:39
Merge branch 'axin' of https://github.com/Thorin215/MiniSQL into axin Star0228 2024/6/1 11:37
Finish_5(non_tested) Star0228 2024/6/1 11:32
Merge branch 'main' of 5.29 into axin Star0228 2024/5/29 9:36
delete wsl debug Star0228 2024/5/28 21:13
xzkz's part update 5.28 Star0228 2024/5/28 18:07
1 2 is done Thorin215 2024/5/28 11:24
add rep Thorin215 2024/5/26 12:09
update 1 Thorin215 2024/5/25 15:57
build new repo for minisql Thorin215 2024/5/13 20:55