数据库实验报告

苏家齐 2021200834

实验完成情况

本次实验共分为4个阶段：存储管理、索引管理、查询执行、并发控制。除第二阶段延期一天外，其余阶段均按时完成，并通过了所有必做的测试点。在选做方面，实验二的任务4选用粗粒度并发完成，实验四完成全部附加实验，其中任务2选用表级锁规避幻读异常。

具体实现算法

实验一：存储管理

数据库里的数据以文件形式存放在磁盘里。当DBMS要从数据库里读取记录时，需要从磁盘中将数据以页（Page）为单位读入到缓冲区，再从缓冲区获取到特定的记录，返回给上层。这个过程经过以下几个类：

1. `DiskManager` 类：这一层负责调用系统提供的 `/usr/include/unistd.h` 中的文件相关接口为数据库创建、删除文件和文件夹，并调用文件读写接口（ `write` `read` 等）从数据文件中以页（Page）为单位读出或写入缓冲区。
2. `Replacer` 类：这一层负责管理缓冲区里的页面，控制页面在缓冲区的固定与淘汰策略。本次实验中使用的是 `LRU` 策略，即当缓冲区满时，淘汰并刷回最远被使用的页面。 `Replacer` 类中维护了一个 `std::list` `LRUList` ，它将按时间顺序维护可被淘汰的页面，并通过 `pin` 和 `unpin` 操作控制页面是否可以被淘汰。
3. `BufferPoolManager` 类：这一层负责管理缓冲区的页面。每一个页面 `Page` 由文件标识符 `fd` 和页面号 `page_no` 唯一确定，并封装成 `Page` 类。 `BufferPoolManager` 维护了一个Page类的数组 `pages_` ，作为缓冲区里的所有帧，通过调用成员 `Replacer replacer_` `DiskManager disk_manager_` 的接口，对缓冲区中的所有frame进行管理，包括空frame（ `std::list<frame_id_t>free_list_` ）和存放了page的frame（ `unordered_map<PageId,frame_id_t>` ），为上层提供了获取页面、刷回页面、新建删除页面等接口。

```
class BufferPoolManager {
  public:
   Page *new_page(PageId *page_id);
   Page *fetch_page(PageId page_id);
   bool unpin_page(PageId page_id, bool is_dirty);
   bool delete_page(PageId page_id);
   bool flush_page(PageId page_id);
   void flush_all_pages(int fd);
  private:
   // 辅助函数
   bool find_victim_page(frame_id_t *frame_id);
   void update_page(Page *page, PageId new_page_id, frame_id_t new_frame_id);
}
```

4. RecordManager 层：主要涉及 RmManager RmFileHandle RmPageHandle RmScan 四个类。这一层负责管理内存中的页面中的记录元组。本实验中，使用定长方式存储记录，每条记录由其字段连接而成，并由 Rid （包括页号和页中槽号）唯一指定。每个页面中最多容纳的记录数是固定的，每条记录通过其在页面中的偏移量访问。
    1. RmManager 类是记录管理器，成员为 disk_manager_ buffer_pool_manager_ ，通过调用二者的接口完成表文件的创建、初始化、删除、关闭
    2. RmFileHandle 类负责一个表文件的管理，成员除了 disk_manager_ buffer_pool_manager_ 以外，还有该记录文件对应的文件标识符，以及包含该表文件元数据的文件头 file_hdr_ （其中包括表中记录大小、页面个数等）。该类核心的功能是负责记录的增删改查，即 insert_record 等函数，其实现为获取页面对应handle、修改bitmap、修改slot、修改元数据。
    3. RmPageHandle 类负责对一个页面进行管理，其中存储所述file的头文件、对应的页面数据page，页面数据包括页面元数据 page_hdr ，标记每个槽位是否被使用的bitmap，以及记录数据slots三个部分。
    4. RmScan 这个类负责实现对表的扫描，其中包含的 rid_ 指向当前访问的数据，提供的 next() 接口会通过在表文件里的遍历，修改 rid_ 使其指向下一个有值的记录。

实验二：索引管理

本实验采用B+树唯一多列索引，B+树以文件形式存放。类似于 RecordManager ，索引管理使用 IxManager 进行索引文件的创建、初始化、打开、关闭，使用 IxFileHandle 对索引进行增删改查，使用 IxNodeHandle 实现对每个结点的管理，使用 IxScan 实现索引树上叶子结点的扫描。

这一阶段主要实现 IxFileHandle 和 IxNodeHandle 里对B+树的查找和维护函数。

1. 查找：从给定的key查找对应的rid，对于结点内部的查找函数 leaf_lookup internal_lookup ，本实验中使用顺序查找的方式找到key对应的rid；对于索引树上的查找函

数 find_leaf_page ，本实验从根节点开始，反复对结点调用 internal_lookup ，沿着B+树逐层向下查找，最终得到需要查找的叶子结点
2. insert：对于结点内部的 insert_pairs 函数，先通过顺序查找找到对应的位置，检查不重复后，将(key,value)对memcpy到指定位置，并将原有的记录往后移动；对于索引树上的插入函数 insert_entry ，在找到对应结点并insert以后，需要检查该结点的键值对数目是否超过上界，如果超过，则需要调用 split 函数分裂成两个节点，并递归修改并检查其父亲结点，直至父亲节点不满、或到达并分裂新的根节点。
3. delete：结点内的 erase_pair 函数思路同 insert ；对于索引树上的删除函数 delete_entry ，在对应结点删除记录以后，需要检查键值对数目是否小于下界，若小于，需要调用 coalesce_or_redistribute 函数，如果从兄弟结点借一个键值对可以保证两个节点不下溢，则调用 redistribute 函数进行重分配，并修改父节点中对应的值；否则，需要将该结点与兄弟节点合并，并递归地修改并检查父节点的下溢情况，直到到达根节点。对于根节点的删除，因为涉及到新的root的指定，因此使用特殊的 adjust_root 函数进行处理。这一部分的两个辅助函数 maintain_parent 和 maintain_child 分别用于修改父节点中的key值，和子节点的parent_node。

除了B+树的维护以外，本阶段还需要完成B+树的并发控制。本实验中，由于时间限制，选择使用粗粒度方法进行实现，即对B+树的三个函数 get_value insert_entry delete_entry ，在函数作用域期间对root_latch_上锁，确保这三个操作的互斥性。

本阶段还需要完成 SmManager::create_index 函数。参考create_table，该函数的实现思路是：生成索引文件的元数据，并相应修改表的元数据，调用 ix_manager_->create_index 创建索引文件；遍历索引建立的表里的所有记录，按照索引列的顺序，拼接每一列的值，作为该条记录的索引key，插入到B+树。

实验三：查询执行

在这一阶段，首先需要实现 SmManager 中对数据库文件夹的创建、打开、关闭，以及对表文件的删除。这一部分使用一些系统命令，创建或删除对应的文件或文件夹，并更新对应的元数据。

当数据库接收到一条语句的时候，它会通过语法分析、语义分析等操作，生成一颗语法树，并使用火山模型生成一个多层的算子查询计划。例如，对于语句 SELECT sname FROM Student WHERE sno = 12 ，其对应的算子从下往上为： seq_scan projection ，即首先在seq_scan层反复在Student表里scan到下一个符合sno=12的记录，然后传递给projection层进行列提取，再返回给最顶层，得到期望形式的记录。

在实验三和实验四中，需要实现的算子有 seq_scan index_scan update insert delete nested_loop projection ，除了最后两个算子以外，前面的算子都位于火山模型的底部，需要通过 RmFileHandle 的RmScan搜索原表；非底部的算子将从其下方的算子，即成员 prev_ 获取处理过的记录，并做进一步的处理。每一个算子需要实现 beginTuple() nextTuple next 三个主要接口和一些辅助接口。

1. beginTuple() ：用于初始化算子。对于scan类的算子，需要用scan_(RmScan or IxScan)使算子指向第一个有值的记录；操作类底层算子（insert delete update）没有beginTuple操作；非底层算子（nested_loop projection）的beginTuple需要让其下层算子调用beginTuple，比如嵌套连接算子，就需要左右两表都指向第一个有值且符合条件的记录

2. `nextTuple()`：用于移动算子指向的记录，适用于scan类算子和非底层算子，需要将算子里的 `rid_` 指向下一个有值且符合条件的记录。需要特别处理的是嵌套连接算子，首先需要将右表算子nextTuple，如果右表到达末尾，需要将右表的rid指向开头，并使左表算子nextTuple，其结束条件是左右算子同时到达末尾。
3. `Next()`：相当于一个“执行”操作。对于scan类算子，执行scan操作，即返回目前算子指向的值；对于操作类算子，该接口负责对目标记录执行 `insert\delete\update` 操作，并且需要相应地更改索引值；对于非底层算子，nested_loop算子需要按照给定字段，连接左右表算子指向的记录，返回给上一层；projection算子需要提取出投影列的值，组合成一条新纪录，返回给上一层。

通过实现以上算子，DBMS可以将一条sql语句处理成算子火山模型，而后通过调用顶层的Next接口，逐层向下，与实验一的存储进行交互，实现对数据库的存储内容的修改。

实验四：并发控制

这一部分需要实现数据库的事务机制，通过实现事务类 `Transaction` 和管理类 `TransactionManager` 进行事务的管理。对于每个事物，有唯一的TransactionID与其匹配，并在算子里维护该事务的写集，记录该事务insert、update、delete的记录，后两者还需要记录其修改前的值，用于abort时的恢复。

`TransactionManager` 负责事务的begin、commit、abort，分别提供了三个接口。

1. `begin`：将新事务加到 `txn_map` 中
2. `commit`：事务的写操作直接执行，因此commit的事务只需要清空写集，并释放锁集里的所有锁，最后将事务日志通过log_manager刷盘（事实上log相关的函数尚未实现），并将事务状态更新为COMMITTED
3. `abort`：abort的事务需要**从后往前**回滚其写集里的insert、delete、update操作，包括恢复其记录及索引数据（具体操作同实验三中算子的Next），并释放锁集里所有锁，事务日志刷盘，并将事务状态更新为ABORTED

之后需要实现锁管理器 `LockManager`，使用2PL no-wait 意向锁策略，需要实现对记录的写锁、读锁，对表的写锁、读锁、IS锁、IX锁、SIX锁。相关的接口如下

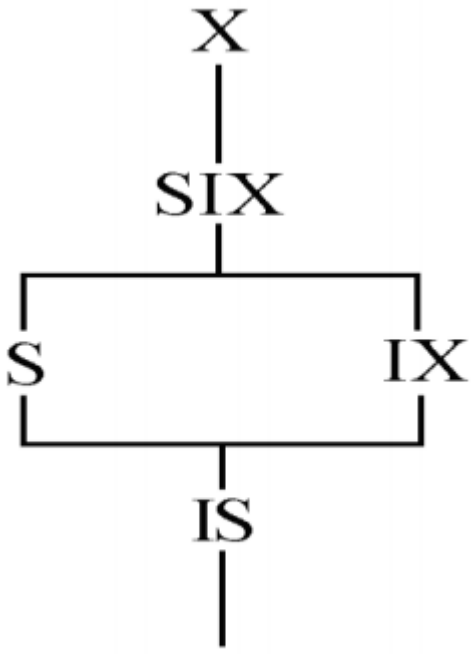
```
class LockManager {
public:
    // 行级锁
    bool lock_shared_on_record(Transaction *txn, const Rid &rid, int tab_fd);
    bool lock_exclusive_on_record(Transaction *txn, const Rid &rid, int tab_fd);
    // 表级锁
    bool lock_shared_on_table(Transaction *txn, int tab_fd);
    bool lock_exclusive_on_table(Transaction *txn, int tab_fd);
    // 意向锁
    bool lock_IS_on_table(Transaction *txn, int tab_fd);
    bool lock_IX_on_table(Transaction *txn, int tab_fd);
    // 解锁
    bool unlock(Transaction *txn, LockDataId lock_data_id);
```

```
private:
    std::unordered_map<LockDataId, LockRequestQueue> lock_table_;
};
```

每个对象（表或行）上的锁由LockDataID唯一确认，`lock_table_`维护了每个对象上被哪些事务上了哪些锁，在每个LockRequestQueue中还包含该对象上最高排他性的锁，以及标记该锁是否有效的granted_（在no-wait下这个标记似乎没有必要，但没有实验验证）。

每一个加锁接口遵循以下步骤：

1. 对全局锁表加锁
2. （2PL）检查并更新事务状态，不允许SHRINKING或已经结束的事务申请锁；其他情况下将事务的状态更新为GROWING
3. 对目标LockDataID对应的requestQueue，遍历一遍，检查两件事
    1. 是否有其它事务的锁，与本事务要加的锁产生冲突，如果产生冲突，立刻throw AbortException，让本事务abort
    2. 是否有本事务的更强或更弱的锁，对于更强的锁，用一个bool进行记录，便于在检查完3.1以后判断是否直接返回；对于更弱的锁，需要记录下指向它的迭代器，便于在检查完3.1以后，直接将那个弱锁升级。锁的强弱关系如下



    在发现同事务的锁之后，没有直接升级或返回。因为假设A有读锁，B也有读锁，这时候A想申请写锁，如果在扫到A的读锁以后就升级返回，就忽视了A写B读之间的冲突，但按no-wait的思想，这里应该是要让A事务abort的。但根据同事务锁的强弱程度，可能可以进行优化，目前还是选择检查完一遍其它事务的锁以后再进行优化和返回
4. 如果检查到同事务更强或同类型的锁，可以直接返回；否则如果是同等级（S和IX）或者更弱的锁，就需要进行相应的升级，并修改groupLockMode

5. 如果没遇到其它事务的锁，也没有本事务的锁，则办法一把新的锁，并在事务的锁集里加入这个锁。

对于不同类型的锁，区别主要在于锁的强弱关系、排斥关系和升级规则，这可以根据锁的相容矩阵和上图的偏序关系决定，在此不赘述。

对于 `unlock` 操作，需要把对应事务的锁从全局锁表里删除，并且扫一遍删除后的requestQueue，更新group_lock_mode。

本实验遵从强2PL，只在事务abort或commit的时候释放锁，因此不会出现在回滚删除操作时，记录没能插入回原位置（因为此时删除事务持有该记录该位置的写锁，其他事务无法向那个位置插入新记录，目前是这个想法）

2PL协议实现

为了实现加锁操作，需要在 `RmFileHandle` 和 `sm_manager` 中按如下方法加锁

接口	锁
Rm get_record	表IS 行S
Rm delete_record/update_record	表IX 行X
Sm drop_table	表X
Sm create_index	表S
Sm drop_index	表S

对于**附加实验**里对幻读的预防，本实验选择在insert_record里加表X锁，可以通过幻读测试的4个点。关于为什么delete_record只需要加IX锁，这是因为当事务A读整个表的时候，在目前的实现里实际是对表里的每一个行都加了一个读锁；而后事务B要删除某个行，需要对这行加些锁，在这里产生的冲突。

最后为了通过幻读测试的第四个点，即索引查询的匹配，对optimizer里的索引规则进行了修改

```
// 目前的索引匹配规则为：完全匹配索引字段，且全部为单点查询，不会自动调整where条件的顺序
// 修改成多点查询，且对index_col_names进行查重
bool Planner::get_index_cols(std::string tab_name, std::vector<Condition> curr_conds,
std::vector<std::string>& index_col_names) {
    index_col_names.clear();
    std::unordered_set<std::string> inserted_cols;
    for(auto& cond: curr_conds) {
        // if(cond.is_rhs_val && cond.op == OP_EQ && cond.lhs_col.tab_name.compare(tab_name)
== 0)
        if(cond.is_rhs_val && cond.lhs_col.tab_name.compare(tab_name) == 0
            && inserted_cols.count(cond.lhs_col.col_name)==0){
            index_col_names.push_back(cond.lhs_col.col_name);
            inserted_cols.insert(cond.lhs_col.col_name);
        }
    }
    TabMeta& tab = sm_manager_->db_.get_table(tab_name);
```

```
    if(tab.is_index(index_col_names)) return true;
    return false;
}
```

创新点与优化技术

在目前的实现里使用的都是最基础的方案，但可以看到存在几个改进的点：

1. 实验二索引管理阶段，在节点内查询时，使用的时顺序查找，但事实上鉴于节点内key的有序性，也可以以二分查找的方式实现；
2. 在index_scan算子里，查询范围设置为第一片叶子至最后一片叶子，算子会将这个区间里的每一个记录都读进来检查条件，但对于范围查询，比如测试点里的id > 2 and id < 10，可以将查询范围进行缩小，把upper缩到id=10的位置，lower缩到id=2的位置。
3. 并发控制阶段，由于对间隙锁还是不太理解，因此选择了表锁来防止幻读，并发效率比较低；在遵循no-wait的系统上，发现abort的频率很高，因此实际场景下no-wait可能不是一个很好的选择。

课程体会和建议

通过一个学期的理论学习，以及自己动手实现一个（半成品）DBMS，我感觉自己对数据库的整个执行逻辑有了更深入的理解，也锻炼了自己的动手能力，尤其是强化了用vscode进行gdb调试的方法（非常受益的一点）。很感谢老师们和助教们的鼓励和帮助，让我有信心自己完成整个实验。