



本科实验报告

课程名称: 数据库系统

姓 名: 黄星尧, 钱梓洋, 姚东雨

学 号: 3230102292, 3230103502, 3230104118

专 业: 计算机科学与技术

指导教师: 陈岭

报告日期: 2025 年 6 月 1 日

Table of Contents

一、 小组总体设计报告	4
1.1 分工 & 协作记录	4
1.2 总体测试	6
二、 个人详细设计报告	11
2.1 Disk and Buffer Pool Manager	11
2.1.1 原理和代码实现	11
2.1.1.1 DiskManager	11
2.1.1.2 BufferPoolManager	13
2.1.2 测试	15
2.1.2.1 ExtentPageAllocationTest	15
2.1.2.2 LRUPerformanceTest	15
2.1.3 Bonus	16
2.1.3.1 测试	17
2.2 Record Manager	18
2.2.1 原理和代码实现	18
2.2.1.1 Record	18
2.2.1.2 TableHeap	18
2.2.2 测试	22
2.3 Index Manager	23
2.3.1 原理和代码实现	23
2.3.1.1 整体构建	23
2.3.1.2 查询操作	24
2.3.1.3 插入操作	25
2.3.1.4 删除操作	27
2.3.1.5 index_iterator	28
2.3.2 测试	29
2.4 Catalog Manager	31
2.4.1 原理和代码实现	31
2.4.1.1 目录元信息	31
2.4.1.2 表和索引的管理	32
2.4.2 测试	44
2.5 Planner and Executor	45
2.5.1 原理	45
2.5.2 代码实现	47
2.5.3 测试	59
2.6 Recovery Manager	61

2.6.1 原理和代码实现	61
2.6.1.1 创建日志	62
2.6.1.2 恢复系统	63
2.6.2 测试	64
2.6.3 思考题	64
2.6.3.1 ARIES 算法	65
2.6.3.2 程序设计	66
2.7 Lock Manager	67
2.7.1 原理	67
2.7.2 代码实现	68
2.7.3 测试	75
2.7.4 思考题	76
2.7.4.1 设计思路	77
2.7.4.2 实现想法	77
三、 对代码框架的一些建议	78

MiniSQL 实验报告

一、 小组总体设计报告

1.1 分工 & 协作记录

以下是我们小组的分工：

- 姚东雨：完成 #1 和 #2（以及 2 的 Bonus），以及 #7 锁管理器部分。
- 黄星尧：完成 #1 的 Bonus, #3 和 #6，并负责 #6 和 #7 思考题的回答。
- 钱梓洋：完成 #4, #5 以及 #7 死锁检测部分，并负责整体报告的撰写。

为了证明小组的每个成员都参与到 MiniSQL 各模块的设计，下面展示了我们项目的 Git 提交历史：

```
* 81ffffcd -(11 hours ago) 完成clock replacer <au12321ua> (HEAD -> NoughtQ, origin/main, origin/HEAD)
|\ \
| * 86e3f70 -(11 hours ago) 完成clock_replacer及其测试 <au12321ua>
| | 6031317 -(3 days ago) 修复删除index不删除实际记录；修复刷库先删文件后释放资源 <au12321ua> (origin/NoughtQ)
| * 429c0a2 -(3 days ago) 修复建立index不插入原有记录的bug；提高计时精度；去除迭代器空白Warning <au12321ua>
| * adbd4f8 -(3 days ago) \#3:去除部分冗余代码 <au12321ua>
|\ \
| * 75c6e1f -(6 days ago) 修复去冗余后出现的问题 <au12321ua> (origin/au)
| * 391235d -(6 days ago) Merge branch 'main' into au, fix one bug in "去除部分冗余代码" <au12321ua>
|\ \
| * 83dead5 -(6 days ago) feat: fix one bug in #4, which hurts the correctness of #5 <NoughtQ>
| * 408251d -(6 days ago) 去除部分冗余代码 <au12321ua>
* | 6f7d114 -(3 days ago) 去除了一条不必要的 error 日志 <Preisthe> (origin/ydy)
* | bb6e8a6 -(3 days ago) Merge main to keep updated <Preisthe>
|\ \
| * | 53720e6 -(4 days ago) fix: (maybe) resolve bugs for not selecting records in existent databases successfully <NoughtQ> (main)
| * | dada186 -(4 days ago) fix: resolve the bug in ExecuteEngine::ExecuteExecfile <NoughtQ>
| * | 5f177fe -(4 days ago) feat: #7 passed all tests <NoughtQ>
| * | 76adee5 -(5 days ago) merge: merge branch ydy(lock manager in #7) into NoughtQ <NoughtQ>
|\ \
| * | | 701ff36 -(5 days ago) feat: finish deadlock detection in #7 <NoughtQ>
* | 3c5923b -(3 days ago) 优化删除、插入，并修改相应测试，关闭了耗时很长的测试 <Preisthe>
|\ //
| * | | 5523de4 -(5 days ago) 完成 lock_manager 并通过测试 <Preisthe>
* | | bae6be6 -(6 days ago) Merge main to keep updated <Preisthe>
|\ \
| * | | db56e58 -(6 days ago) merge: merge branch NoughtQ into main for updating #4 and #5 <NoughtQ>
|\ \
| * | | 0d25bd4 -(6 days ago) chore: add comments for execute_engine.cpp <NoughtQ>
| * | | 1b8b18f -(6 days ago) fix: resolve the bug in #5 (passed almost all tests in #9) <NoughtQ>
| * | | 4cf964d -(6 days ago) fix: fix one bug in #4, which hurts the correctness of #5 <NoughtQ>
|\ \
| * | | 6355b2a -(6 days ago) fix: fix one bug in #4, which hurts the correctness of #5 <NoughtQ>
|\ //
| * | | 74f7a30 -(6 days ago) feat: add extended catalog test and fix some bugs in #4 <NoughtQ>
|\ \
| * | | ce36f2f -(8 days ago) merge: merge #6 into branch NoughtQ <NoughtQ>
|\ \
| * | | a953b35 -(9 days ago) 合并 #6 <au12321ua>
|\ \
| | * 7d42ed7 -(9 days ago) #6 测试通过 <au12321ua>
| * | | 382a835 -(8 days ago) fix: resolve the bug in CatalogManager::LoadTable <NoughtQ>
|\ //
| * | | 85fefab -(9 days ago) merge: update b_plus_tree.cpp from branch au <NoughtQ>
|\ \
| | * 60b18c0 -(9 days ago) 修复删除单个孩子的root后重复unpin问题，整理涉及header的代码 <au12321ua>
| * | | 44b073f -(9 days ago) docs: add comments for catalog.cpp and execute_engine.cpp <NoughtQ>
|\ /
| * | | 1183547 -(11 days ago) chore: remove helper codes for #4 test <NoughtQ>
| * | | 9eaba88 -(11 days ago) feat: pass all tests of #4 <NoughtQ>
| * | | 477a8b3 -(11 days ago) merge: merge the patch about RootPageId in #3 into branch NoughtQ <NoughtQ>
```

```

| \
| * e22fc8d -(11 days ago) 修复RootPageId更新问题 <au12321ua>
| * | 3fa2405 -(12 days ago) Merge branch 'main' into NoughtQ <NoughtQ>
| \
| * | | c2c8737 -(12 days ago) feat: pass 2/3 of catalog test <NoughtQ>
| * | | f181db1 -(13 days ago) merge: merge part 1&2 to branch NoughtQ <NoughtQ>
| \
| * | | | 8d993fc -(13 days ago) feat: finish coding of part5, but not yet test the code. <NoughtQ>
| * | | | 8a8d702 -(4 weeks ago) feat: finish catalog manager, but only pass 1st unit test because of dependences of other components <NoughtQ>
| * | | | 9403941 -(9 days ago) 修复 merge 导致的重复 #include <Preisthe>
| * | | | 154277a -(9 days ago) Merge main into ydy <Preisthe>
| \
| | \
| | | 3f7c477 -(13 days ago) 合并 #3 <au12321ua>
| \
| | \
| | | 5175045 -(13 days ago) Merge branch 'ydy' into au <au12321ua>
| \
| * | | 6f37ac8 -(13 days ago) 增加了将树删空的测试 <au12321ua>
| * | | d52d2c0 -(13 days ago) 修复了内部节点删除时遇到的一些问题 <au12321ua>
| * | | 63620a1 -(2 weeks ago) 修复了删除时挪用操作后未解除 sibling 占用的 bug <au12321ua>
| * | | ed6c6b9 -(2 weeks ago) 修复了 delete 部分的 bug, 重写 End(), 通过 test/index 中的三个测试 <au12321ua>
| * | | ca7b8c3 -(2 weeks ago) Insert 部分修了一些 bug <au12321ua>
| * | | cabf28c -(2 weeks ago) Merge branch 'ydy' into au <au12321ua>
| \
| * | | | 543d754 -(3 weeks ago) 修复 delete 前没有 unpin 的问题 <xinyao huang>
| * | | | c0bb92e -(3 weeks ago) Merge branch 'ydy' into au <au12321ua>
| \
| * | | | ea25cff -(3 weeks ago) 修了一些 bug <au12321ua>
| * | | | d99b208 -(4 weeks ago) iterator finish (?) <xinyao huang>
| * | | | 673195f -(4 weeks ago) delete finish (?) <xinyao huang>
| * | | | c967025 -(4 weeks ago) insert finish (?) <xinyao huang>
| * | | | f9e8f2c -(4 weeks ago) page finish (?) <xinyao huang>
| \
| * | | | 0684754 -(13 days ago) 合并 #1 #2 部分, 并提供接口注释 <Preisthe>
| \
| * | | | 9c323ff -(13 days ago) 修改了析构实例的顺序 <Preisthe>
| \
| * | | | 0af687c -(2 weeks ago) 更新了一些检查, 增加了新页的数据清空, 新增一个接口注释 <Preisthe>
| \
| * | | | 22377cc -(2 weeks ago) 修复了页的 is_dirty 被覆盖的问题, 更新了一些测试 <Preisthe>
| * | | | 4742ab4 -(2 weeks ago) 增加了 diskmanager 和 tableHeap 的测试, 解决了 tableHeap 迭代器不能跨页的问题 <Preisthe>
| * | | | 2ab116d -(2 weeks ago) 修改了一堆 bug <Preisthe>
| * | | | 1d36b7c -(3 weeks ago) 添加了一些测试, 修改了部分 bug <Preisthe>
| * | | | eef18b6 -(3 weeks ago) [bug fix] 给 table_heap 添加构造函数 <Preisthe>
| \
| * | | | e5b5194 -(3 weeks ago) add unordered_map <Preisthe>
| * | | | 6091b6d -(3 weeks ago) fix disk and buffer pool manager bugs <Preisthe>
| * | | | b12a101 -(3 weeks ago) 完成 table heap <Preisthe>
| * | | | 8ac1d3f -(3 weeks ago) disk and buffer pool manager 和 record manager 中的 serialization 完成 <Preisthe>
| \
| * | | | cce0909 -(5 weeks ago) Init: Adding a framework <au12321ua>

```

Figure 2: Git Commit History

注

- @Preisthe: 姚东雨
- @au12321ua: 黄星尧
- @NoughtQ: 钱梓洋

1.2 总体测试

运行 `minysql_test` 程序，得到的总体测试结果如下：

```
[=====] Running 40 tests from 12 test suites.
[-----] Global test environment set-up.
[-----] 2 tests from BufferPoolManagerTest
[RUN    ] BufferPoolManagerTest.BinaryDataTest
[OK    ] BufferPoolManagerTest.BinaryDataTest (72 ms)
[RUN    ] BufferPoolManagerTest.LRUPerformanceTest
Filling buffer pool...
Accessing the whole buffer pool...
Accessing some pages that have been replaced...
Accessing some recently accessed pages...
Accessing some replaced pages...
Deleting some pages...
Creating new pages...
Verifying frequent accessed pages...
[OK    ] BufferPoolManagerTest.LRUPerformanceTest (16921 ms)
[-----] 2 tests from BufferPoolManagerTest (16993 ms total)

[-----] 1 test from CLOCKReplacerTest
[RUN    ] CLOCKReplacerTest.SampleTest
[OK    ] CLOCKReplacerTest.SampleTest (0 ms)
[-----] 1 test from CLOCKReplacerTest (0 ms total)

[-----] 1 test from LRUReplacerTest
[RUN    ] LRUReplacerTest.SampleTest
[OK    ] LRUReplacerTest.SampleTest (0 ms)
[-----] 1 test from LRUReplacerTest (0 ms total)

[-----] 5 tests from CatalogExtendedTest
[RUN    ] CatalogExtendedTest.DropTableAndIndexTest
[OK    ] CatalogExtendedTest.DropTableAndIndexTest (149 ms)
[RUN    ] CatalogExtendedTest.BatchRetrievalTest
[OK    ] CatalogExtendedTest.BatchRetrievalTest (177 ms)
[RUN    ] CatalogExtendedTest.GetTableByIdTest
[OK    ] CatalogExtendedTest.GetTableByIdTest (145 ms)
[RUN    ] CatalogExtendedTest.ErrorHandlingTest
[OK    ] CatalogExtendedTest.ErrorHandlingTest (154 ms)
[RUN    ] CatalogExtendedTest.ComplexScenarioAndPersistenceTest
[OK    ] CatalogExtendedTest.ComplexScenarioAndPersistenceTest
(297 ms)
```

```
[-----] 5 tests from CatalogExtendedTest (925 ms total)

[-----] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[ OK       ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[ OK       ] CatalogTest.CatalogTableTest (278 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[ OK       ] CatalogTest.CatalogIndexTest (284 ms)
[-----] 3 tests from CatalogTest (563 ms total)

[-----] 10 tests from LockManagerTest
[ RUN      ] LockManagerTest.SLockInReadUncommittedTest
[ OK       ] LockManagerTest.SLockInReadUncommittedTest (0 ms)
[ RUN      ] LockManagerTest.TwoPhaseLockingTest
[ OK       ] LockManagerTest.TwoPhaseLockingTest (0 ms)
[ RUN      ] LockManagerTest.UpgradeLockInShrinkingPhase
[ OK       ] LockManagerTest.UpgradeLockInShrinkingPhase (0 ms)
[ RUN      ] LockManagerTest.UpgradeConflictTest
[ OK       ] LockManagerTest.UpgradeConflictTest (105 ms)
[ RUN      ] LockManagerTest.UpgradeTest
[ OK       ] LockManagerTest.UpgradeTest (0 ms)
[ RUN      ] LockManagerTest.UpgradeAfterAbortTest
[ OK       ] LockManagerTest.UpgradeAfterAbortTest (105 ms)
[ RUN      ] LockManagerTest.BasicCycleTest1
[ OK       ] LockManagerTest.BasicCycleTest1 (0 ms)
[ RUN      ] LockManagerTest.BasicCycleTest2
[ OK       ] LockManagerTest.BasicCycleTest2 (0 ms)
[ RUN      ] LockManagerTest.DeadlockDetectionTest1
[ OK       ] LockManagerTest.DeadlockDetectionTest1 (1509 ms)
[ RUN      ] LockManagerTest.DeadlockDetectionTest2
[ OK       ] LockManagerTest.DeadlockDetectionTest2 (1110 ms)
[-----] 10 tests from LockManagerTest (2831 ms total)

[-----] 4 tests from ExecutorTest
[ RUN      ] ExecutorTest.SimpleSeqScanTest
[ OK       ] ExecutorTest.SimpleSeqScanTest (235 ms)
[ RUN      ] ExecutorTest.SimpleDeleteTest
[ OK       ] ExecutorTest.SimpleDeleteTest (329 ms)
[ RUN      ] ExecutorTest.SimpleRawInsertTest
[ OK       ] ExecutorTest.SimpleRawInsertTest (217 ms)
[ RUN      ] ExecutorTest.SimpleUpdateTest
```

```
[       OK ] ExecutorTest.SimpleUpdateTest (227 ms)
[-----] 4 tests from ExecutorTest (1010 ms total)

[-----] 4 tests from BPlusTreeTests
[ RUN     ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (0 ms)
[ RUN     ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (141 ms)
[ RUN     ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (7829 ms)
[ RUN     ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (145 ms)
[-----] 4 tests from BPlusTreeTests (8117 ms total)

[-----] 1 test from PageTests
[ RUN     ] PageTests.IndexRootsPageTest
[       OK ] PageTests.IndexRootsPageTest (0 ms)
[-----] 1 test from PageTests (0 ms total)

[-----] 2 tests from TupleTest
[ RUN     ] TupleTest.FieldSerializeDeserializeTest
[       OK ] TupleTest.FieldSerializeDeserializeTest (0 ms)
[ RUN     ] TupleTest.RowTest
[       OK ] TupleTest.RowTest (0 ms)
[-----] 2 tests from TupleTest (0 ms total)

[-----] 1 test from RecoveryManagerTest
[ RUN     ] RecoveryManagerTest.RecoveryTest
[       OK ] RecoveryManagerTest.RecoveryTest (0 ms)
[-----] 1 test from RecoveryManagerTest (0 ms total)

[-----] 6 tests from TableHeapTest
[ RUN     ] TableHeapTest.HeapPerformanceTest
[       OK ] TableHeapTest.HeapPerformanceTest (298 ms)
[ RUN     ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (691 ms)
[ RUN     ] TableHeapTest.TableHeapDeleteTest
[       OK ] TableHeapTest.TableHeapDeleteTest (131 ms)
[ RUN     ] TableHeapTest.TableHeapUpdateTest
[       OK ] TableHeapTest.TableHeapUpdateTest (154 ms)
[ RUN     ] TableHeapTest.TableHeapMassiveTest
```

1

```

1.5
2
3
4
5
6
7
8
[      OK ] TableHeapTest.TableHeapMassiveTest (148 ms)
[ RUN      ] TableHeapTest.TableIteratorTest
1
2
3
4
[      OK ] TableHeapTest.TableIteratorTest (127 ms)
[-----] 6 tests from TableHeapTest (1552 ms total)

[-----] Global test environment tear-down
[=====] 40 tests from 12 test suites ran. (31992 ms total)
[ PASSED ] 40 tests.

```

此外，我们的程序也能在主程序上成功运行，并按照实验文档给出的“#9 MiniSQL 项目验收流程”，基本上能通过所有的测试，并且也已经通过了助教的验收。

但在验收的时候，我们遇到了一个小问题：程序没法顺利完成“先删记录，再删索引，最后再插入删去的记录”的流程。然而，我们在验收后尝试了各种查询组合，都没能复现这个问题。但在探索过程中，我们发现了在 `drop index` 时没能删除实际的索引数据结构。但这一问题实际上不会影响到插入，因为 `index_info` 已经被清除。修复这个问题后，多次尝试同样没有再出现验收时的问题。在此基础上，我们认为这个 bug 是小概率发生的事件。

```

minysql > delete from account where name = "name45678";
Query OK, 1 row affected(278.9500 ms).
minysql > drop index idx01;
minysql > insert into account values(12345678, "name45678", 114514);
Query OK, 1 row affected(0.5060 ms).
minysql > █

```

另外，删除非空的数据库是出现大量警告：Warning:Attempting to write free page。这个问题实际上并不是我们代码的问题，而是框架在 `ExecuteEngine::ExecuteDropDatabase(*ast, *context)` 中存在的逻辑问题。

代码先删除文件，再进行 `DBStorageEngine` 对象的析构。析构函数中又调用了 `CatalogManager` 的析构函数，试图将脏页写回已删除的文件中，造成了警告。

要想让程序不报错，只需对调 `DBStorageEngine` 的析构函数和删除文件的顺序即可。结果如下所示：

```
minisql > drop database db0;
minisql > show databases;
+-----+
| Database |
+-----+
| db2      |
| db1      |
+-----+
minisql > █
```

二、个人详细设计报告

2.1 Disk and Buffer Pool Manager

2.1.1 原理和代码实现

该模块分为两部分：Disk Manager 和 Buffer Pool Manager。

2.1.1.1 DiskManager

该模块的基本思想是，用位图(bitmap)管理分区，用 disk_file_meta_page 管理多个分区，以此来达到管理大量磁盘空间的目的，结构如下图所示：

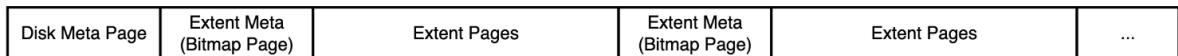


Figure 5: DiskManager 结构

2.1.1.1.1 单个分区的页面管理

其中，bitmap 设计为用二进制位表示一个页是否被使用，因此大小为 4KB 的 bitmap 可以管理 32K 个页，这就是一个分区中页的数量。

当分配页时，系统需要从所有二进制位中找到一个空闲的位并进行分配，为了提高效率，可以维护一个 next_free_page_ 变量，每次分配新页时，分配该页，然后从该页开始往后查找空闲页。下面是实现代码：

src/page/bitmap_page.cpp

```
1 template <size_t PageSize>
2     bool BitmapPage<PageSize>::AllocatePage(uint32_t
3         &page_offset) {
4     if (next_free_page_ == INVALID_PAGE) {
5         return false;
6     }
7     if (page_allocated_ == GetMaxSupportedSize()) {
8         LOG(ERROR) << "No more space in the bitmap page";
9         return false;
10    }
11    page_offset = next_free_page_; // 分配下一个可用的页面
12
13    /* 更新位图 */
14    uint32_t byte_index = page_offset / 8;
15    uint8_t bit_index = page_offset % 8;
16    bytes[byte_index] |= (1 << bit_index);
17    page_allocated_++; // 增加已分配的页面数
18
19    /* 更新下一个可用的页面 */
```

```

19     uint32_t start = next_free_page_;
20         next_free_page_ = (next_free_page_ + 1) %
21     GetMaxSupportedSize(); // 循环使用位图 (circular bitmap)
22     while(next_free_page_ != start){
23         if(IsPageFree(next_free_page_)){
24             break;
25         }
26         next_free_page_ = (next_free_page_ + 1) %
27     GetMaxSupportedSize(); // 循环使用位图 (circular bitmap)
28     }
29     if(next_free_page_ == start){
30         next_free_page_ = INVALID_PAGE; // 没有可用的页面
31     }
32     return true;
33 }
```

当分区很满时，该方法效率可能变低，比如最坏情况下，下一个空闲页刚好是当前页的前一个页，此时需要遍历整个位图。但我在后续测试中发现，分配页时候最大的开销其实是在 bitmap 页的读取上，当整个 bitmap 已经被读入内存时，其实遍历的开销几乎可以忽略不计。

基于这个分配策略，也可以轻松完成 DeAllocatePage 和 IsPageFree 方法。

2.1.1.1.2 多个分区的管理

我们通过在 disk_file_meta_page 中记录每个分区以分配的页的数量，对多个分区进行高效管理，并封装成接口供上层使用。同时，由于 bitmap 页的存在，还需要建立物理页号和逻辑页号的映射关系，以便上层使用。

物理页号	0	1	2	3	4	5	6	...
职责	磁盘元数据	位图页	数据页	数据页	数据页	位图页	数据页	
逻辑页号	/	/	0	1	2		3	

Figure 6: DiskManager 结构

在分区的管理上，我们采用了如下策略：若当前分区已满，那就直接新开一个分区进行分配，而不寻找之前的分区是否有空闲页。

这样做是考虑到之前的分区虽然可能有空闲，但一般不会太多，分区依旧处于快满的状态，分配的效率可能会很低，而新分区就可以无脑分配了。

src/storage/disk_manager.cpp

```

1 page_id_t DiskManager::AllocatePage() {
2     DiskFileMetaPage *meta_page = reinterpret_cast<DiskFileMetaPage *>(meta_data_);
3     uint32_t &num_extents = meta_page->num_extents_;
4     if (num_extents == 0 || meta_page->GetExtentUsedPage(num_extents - 1) == BITMAP_SIZE) {
5         // 开辟新的 extent
6         num_extents++;
7     }
8     if (num_extents > (PAGE_SIZE - 8) / 4) {
9         LOG(ERROR) << "No more space for new page.";
10    return INVALID_PAGE_ID;
11 }
12 /* 在新 extent 中分配页 */
13 char bitmap_page_data[PAGE_SIZE] = {0}; // 存储位图的内存
14 BitmapPage<PAGE_SIZE> *bitmap_page = reinterpret_cast<BitmapPage<PAGE_SIZE *>>(bitmap_page_data);
15 ReadPhysicalPage((num_extents - 1) * (BITMAP_SIZE + 1) + 1,
16 bitmap_page_data); // 读取位图页
17
18 uint32_t page_offset; // 分区中的页偏移
19 bool success = bitmap_page->AllocatePage(page_offset); // 分配页
20 ASSERT(success, "Failed to allocate page.");
21 WritePhysicalPage((num_extents - 1) * (BITMAP_SIZE + 1) + 1,
22 bitmap_page_data); // 写入位图页
23
24 // 更新元数据
25 meta_page->num_allocated_pages_++;
26 meta_page->extent_used_page_[num_extents - 1]++;
27
28 return (num_extents - 1) * BITMAP_SIZE + page_offset;
29 }
```

然后再实现释放页和查询页是否空闲的方法即可。

2.1.1.2 BufferPoolManager

该模块的作用是将对磁盘页的读取和管理封装起来，同时维护一个缓冲池以提高读取效率。

对于页面的分配、回收与读取，实验框架里给出了较为详细的指导，因此照着实现即可，主要需要注意的是页在内存中和在磁盘中时不同的处理策略。

然后就是缓冲池的替换逻辑，这里采用了 LRU 算法，用一个双向链表表示 LRU 队列，再用一个哈希表记录页在链表中的位置实现快速访问。需要注意的是在内

存中我们就不能用 page_id 识别页了，而是这页数据在缓冲池中的位置，我们把它抽象为 frame_id，用于管理内存中的页。因此 LRU 队列中存储的是 frame_id。

src/include/buffer/lru_replacer.h

```
1 private:
2     // add your own private member variables here
3     list<frame_id_t> lru_list_; // 双向链表存储 LRU 队列
4     unordered_map<frame_id_t, list<frame_id_t>::iterator>
5     lru_map_; // 哈希表存储 LRU 队列中的元素及其对应的迭代器
6     size_t max_size_; // 最大容量
```

还需要注意 buffer pool 在执行 UnpinPage 时接收了一个 is_dirty 参数用于维护 Page 的 is_dirty_ 状态，但不能简单赋值，因为 UnpinPage 可能被多次调用，只有在 is_dirty 参数为 true 时才需要更新 is_dirty_ 状态。可以简化为 \Leftarrow 运算符。

src/buffer/buffer_pool_manager.cpp

```
1 bool BufferPoolManager::UnpinPage(page_id_t page_id, bool
2 is_dirty) {
3     auto it = page_table_.find(page_id);
4     if(it == page_table_.end()) {
5         LOG(ERROR) << "Page not in buffer pool: " << page_id
6 << endl;
7         return false;
8     }
9     frame_id_t frame_id = it->second;
10    Page *page = &pages_[frame_id];
11    // LOG(INFO) << "Unpin page: " << page_id << ", pin count:
12    " << page->pin_count_ << endl;
13    if(page->pin_count_ == 0) {
14        LOG(ERROR) << "Unable to unpin page " << page_id << ":";
15        pin_count = " << page->pin_count_ << endl;
16        return false;
17    }
18    page->pin_count_--;
19    if(page->pin_count_ == 0) {
20        replacer_->Unpin(frame_id);
21    }
22    page->is_dirty_  $\Leftarrow$  is_dirty;
23    return true;
24 }
```

2.1.2 测试

针对我们自己的实现逻辑，我给 DiskManager 和 BufferPoolManager 补充了两个测试，用于测试系统在大规模数据下的正确性。

2.1.2.1 ExtentPageAllocationTest

流程如下：

- 先填满第一个分区
- 释放两块连续区域：[100-199], [300-399]
- 重新分配，验证分配的页仍在第一个分区的这两块区域
- 继续分配，验证新的页在第二个分区
- 释放第一个分区的 100 个页面后，继续分配 100 个新页，验证这些页仍然分配在第二个分区

2.1.2.2 LRUPerformanceTest

流程如下：

- 首先分配两倍于缓冲池大小的页并写入测试数据
- Pin 住缓冲池一半数量的页，其余 Unpin
- 由于 lru 具有确定性，我们可以推算出缓冲池中应该存在的页
- 频繁访问这些页，不管有没有 Pin
- 再访问所有不在缓冲池中的页并验证数据
- 访问刚读进来的页和被 Pin 的页
- 访问被替换掉的页
- 最后测试页面删除和重新创建

这样，通过分析各个函数执行的用时，我们可以验证缓冲区是否按照预期工作。（代码太长了）

测试结果如下：

```
W20250531 21:38:40.146737 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146742 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146747 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146752 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146757 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146762 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146767 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146772 9359781 bitmap_page.cpp:54] Page is already free
W20250531 21:38:40.146777 9359781 bitmap_page.cpp:54] Page is already free
[ OK ] DiskManagerTest.BitMapPageTest (42 ms)
[ RUN ] DiskManagerTest.FreePageAllocationTest (357028 ms)
[ RUN ] DiskManagerTest.ExtentPageAllocationTest (196107 ms)
[ OK ] DiskManagerTest.ExtentPageAllocationTest (196107 ms)
[ ----- ] 3 tests from DiskManagerTest (553180 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test suite ran. (553180 ms total)
[ PASSED ] 3 tests.

[ OK ] LRUReplacerTest (42 ms)
[ RUN ] LRUReplacerTest.SampleTest
W20250531 21:41:20.550640 9390641 lru_replacer.cpp:45] frame_id already exists: 1
W20250531 21:41:20.551072 9390641 lru_replacer.cpp:32] Pin frame_id not found: 3
[ OK ] LRUReplacerTest.SampleTest (0 ms)
[ ----- ] 1 test from LRUReplacerTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 1 test.

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (32197 ms total)
[ PASSED ] 2 tests.

[-----] Global test environment tear-down
[=====] 2 tests from BufferPoolManagerTest (32081 ms)
[-----] 2 tests from BufferPoolManagerTest

[ RUN ] BufferPoolManagerTest.BinaryDataTest (116 ms)
[ RUN ] BufferPoolManagerTest.LRUPerformanceTest (32081 ms)
[-----] 2 tests from BufferPoolManagerTest (32081 ms)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (32197 ms total)
[ PASSED ] 2 tests.
```

Figure 7: buffer_pool_manager_test 以及 lru_replacer_test 的测试结果

2.1.3 Bonus

Clock 算法是一种 LRU 算法的良好近似。其想法很简单：模拟时钟指针一圈一圈遍历，直到找到第一个符合替换的位置。算法细节如下：

- **初始化：**创建一个数组 `frames` 来跟踪当前内存中的页面，并创建一个布尔数组 `second_chance` 来跟踪该页面自上次替换以来是否被访问过（即该页面是否值得获得第二次机会），以及一个变量指针来跟踪替换的目标。
- **寻址并插入：**
 - 开始遍历数组 `arr`。如果页面已存在，只需将 `second_chance` 中其对应的元素设置为 `true` 并返回。
 - 如果页面不存在，则检查指针指向的空间是否为空（表示缓存尚未满）。如果是，我们将元素放入该空间并移动指针至下一位；否则，我们将逐个遍历数组 `arr`（循环使用指针的值），将所有对应的 `second_chance` 元素标记为 `false`，直到找到一个已经为 `false` 的页面。这就是最适合替换的页面，因此我们执行替换操作并返回。
 - 便于理解，可以参考下图中的例子：

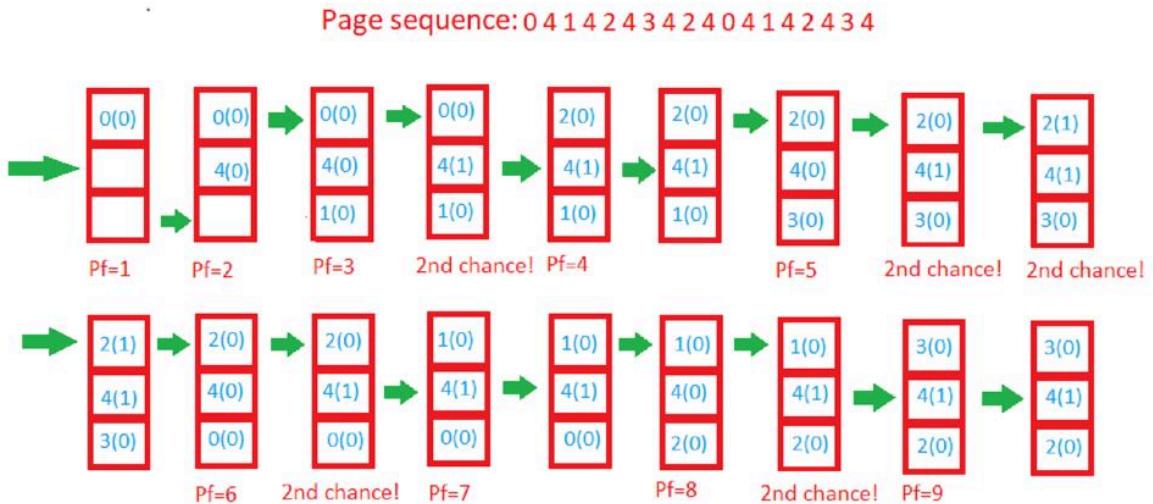


Figure 8: Clock 算法插入示例

但在我们的框架中，寻址并插入被拆分成了 `Victim()`、`Pin()` 与 `Unpin()`，所以需要对算法进行一定调整。另外由于储存数据的不同，我创建了一个 `vector<pair<bool, bool>>` 用来跟踪当前内存中的页面，并通过 `map<size_t, frame_id_t>` 构建起索引与帧数的映射。调整后的各函数方法如下：

- `Victim()`：指针循环遍历：如果所在位无效，则移动到下一位；若有效但 ref bit 为 1，则将本位 ref bit 修改为 0，然后移动到下一位；若所在位有效且 ref bit 为 0，则输出并删除该位置上的内容，不移动指针
- `Pin()`：若存在，则删除目标帧，不移动指针

- `Unpin()`: 与“寻址并插入”设计一致，但实际使用时，上层不会在已满或者已存在的时候调用这一方法，所以实际上存在一定设计冗余

2.1.3.1 测试

采用与 LRU Replacer 相似的测试程序，不过根据算法的不同对判断内容作出修改：

```
test/buffer/lru_replacer_test.cpp

1 TEST(CLOCKReplacerTest, SampleTest) {
2     CLOCKReplacer clock_replacer(7);
3
4     // Scenario: unpin six elements, i.e. add them to the
5     // replacer.
6     clock_replacer.Unpin(1);
7     clock_replacer.Unpin(2);
8     clock_replacer.Unpin(3);
9     clock_replacer.Unpin(4);
10    clock_replacer.Unpin(5);
11    clock_replacer.Unpin(6);
12    clock_replacer.Unpin(1);
13    EXPECT_EQ(6, clock_replacer.Size());
14
15    // Scenario: get three victims from the clock.
16    int value;
17    clock_replacer.Victim(&value);
18    EXPECT_EQ(2, value);
19    clock_replacer.Victim(&value);
20    EXPECT_EQ(3, value);
21    clock_replacer.Victim(&value);
22    EXPECT_EQ(4, value);
23
24    // Scenario: pin elements in the replacer.
25    // Note that 3 has already been victimized, so pinning 3
26    // should have no effect.
27    clock_replacer.Pin(4);
28    clock_replacer.Pin(5);
29    EXPECT_EQ(2, clock_replacer.Size());
30
31    // Scenario: unpin 5, the hand will point to 6 now
32    clock_replacer.Unpin(5);
33
34    // Scenario: continue looking for victims. We expect these
35    // victims.
36    clock_replacer.Victim(&value);
37    EXPECT_EQ(6, value);
38    clock_replacer.Victim(&value);
```

```
36     EXPECT_EQ(1, value);
37     clock_replacer.Victim(&value);
38     EXPECT_EQ(5, value);
39 }
```

测试结果如图：

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[-----] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from CLOCKReplacerTest
[ RUN   ] CLOCKReplacerTest.SampleTest
[ OK    ] CLOCKReplacerTest.SampleTest (0 ms)
[-----] 1 test from CLOCKReplacerTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 1 test.
[Inferior 1 (process 29092) exited normally]
```

Figure 9: 时钟测试结果

2.2 Record Manager

2.2.1 原理和代码实现

本模块主要实现记录（record）的序列化与反序列化，并在此基础上实现 TableHeap。

2.2.1.1 Record

对于相关的四个类，我认为它们的关系如下：

- Column 相当于标明一个列的名字和属性
- Schema 是列的集合，也就是一个表的结构

上面这两表示抽象的结构，下面的存具体数据

- Field 就是某行某列的一个具体值
- Row 是一行，也就是好几个 Field 的集合

处理一个表的时候，上下两部分的信息要吻合。

序列化与反序列化的操作比较简单，无脑塞数据就行了。用 null bitmap 序列化 Row 对象时，只要先根据 Row 对象里各个 Field 的值构造 bitmap（用 32 位的 int 表示），然后序列化即可；反序列化时，先反序列化出 bitmap，然后根据 bitmap 构造出相应的 Field 对象。

2.2.1.2 TableHeap

这里用来存储纯数据，数据无序。一个 TableHeap 的 Schema 是固定的。

2.2.1.2.1 插入

最朴素的做法是从头开始遍历，找到第一个空闲的位置插入。但这样做的复杂度是 $O(n)$ ，效率较低。特别是当整个表超出缓冲池的容量时，由于 lru 的替换策略，每插入一条数据就要把所有页从磁盘中读一遍，效率会更低。

因此，我们完成了 **Bonus**，利用额外的信息来加速插入。具体来说，我们使用一个哈希表来记录表中每个页的剩余空间，每次插入时直接遍历哈希表找一个剩余空间足够的页插入即可。虽然这样不能保证数据从前往后插入（因为哈希表的键值无序），但并不影响，因为堆表本身就是无序的。

```
src/include/storage/table_heap.h
```

```
1 unordered_map<page_id_t, uint32_t> free_space_;
```

优化后的插入代码如下：

```
src/storage/table_heap.cpp
```

```
1 bool TableHeap::InsertTuple(Row &row, Txn *txn) {
2     if (first_page_id_ == INVALID_PAGE_ID) {
3         LOG(ERROR) << "Failed to insert tuple: table is empty"
4         << std::endl;
5     }
6     if (row.GetSerializedSize(schema_) >
7         TablePage::SIZE_MAX_ROW) {
8         LOG(ERROR) << "Failed to insert tuple: tuple is too
9         large" << std::endl;
10    return false;
11 }
12 // 寻找合适的页面
13 uint32_t size = row.GetSerializedSize(schema_) + 8;
14 page_id_t page_id = INVALID_PAGE_ID;
15 auto it = free_space_.begin();
16 for (; it != free_space_.end(); ++it) {
17     if (it->second >= size) {
18         page_id = it->first;
19         break;
20     }
21 }
22 if (page_id != INVALID_PAGE_ID) {
23     // 在找到的页面中插入元组
24     auto page = reinterpret_cast<TablePage
25     *>(buffer_pool_manager_->FetchPage(page_id));
26     if (page == nullptr) {
```

```

23         LOG(ERROR) << "Failed to fetch page when insert"
24     << std::endl;
25     return false;
26 }
27 page->WLatch();
28 // 插入并更新 free space
29 bool insert_success = page->InsertTuple(row, schema_,
30 txn, lock_manager_, log_manager_);
31 it->second = page->GetFreeSpaceRemaining();
32 page->WUnlatch();
33 buffer_pool_manager_->UnpinPage(page->GetTablePageId(),
34 true);
35 if (!insert_success) {
36     assert(false);
37     LOG(ERROR) << "Unexpected error while insert"
38 << std::endl;
39     }
40     return insert_success;
41 } else {
42     // 如果所有现有页面都已满, 创建新页面
43     auto page = reinterpret_cast<TablePage
44 *>(buffer_pool_manager_->FetchPage(last_page_id_));
45     if (page == nullptr) {
46         LOG(ERROR) << "Failed to fetch page when insert"
47 << std::endl;
48         return false;
49     }
50     // 确保是最后一页
51     auto next_page_id = page->GetNextPageId();
52     if (next_page_id != INVALID_PAGE_ID) {
53         buffer_pool_manager_->UnpinPage(page-
54 >GetTablePageId(), false);
55         LOG(ERROR) << "Unexpected error while insert"
56 << std::endl;
57         return false;
58     }
59     // 创建新页面
60     page_id_t new_page_id;
61     auto new_page = reinterpret_cast<TablePage
62 *>(buffer_pool_manager_->NewPage(new_page_id));
63     if (new_page == nullptr) {
64         LOG(ERROR) << "Failed to create new page while
65 insert" << std::endl;
66         return false;
67     }

```

```

58         new_page->Init(new_page_id, page->GetTablePageId(),
59         log_manager_, txn);
59         new_page->SetNextPageId(INVALID_PAGE_ID);
60         page->SetNextPageId(new_page_id);
61         buffer_pool_manager_->UnpinPage(page->GetTablePageId(),
61         true);
62
63         new_page->WLatch();
64         bool insert_success = new_page->InsertTuple(row,
64         schema_, txn, lock_manager_, log_manager_);
65         new_page->WUnlatch();
66         if (!insert_success) {
67             LOG(ERROR) << "Unexpected error while insert"
67             << std::endl;
68         }
69         free_space_.emplace(new_page_id, new_page-
69         >GetFreeSpaceRemaining());
70         last_page_id_ = new_page_id;
71
72         buffer_pool_manager_->UnpinPage(new_page_id, true);
73         return insert_success;
74     }
75 }
```

2.2.1.2.2 更新

对于更新，需要考虑的问题是记录原来存储的位置可能放不下新的数据。TablePage 类提供了原地更新的接口，但没有提供错误原因的反馈，于是我加了一个 valid 参数用于反馈错误原因，即反馈当前页更新失败是由于空间不足还是其他错误。

src/page/table_page.cpp

```

1  bool TablePage::UpdateTuple(Row &new_row, Row *old_row, Schema
2  *schema, bool &valid, Txn *txn, LockManager *lock_manager,
2  LogManager *log_manager) {
3      ASSERT(old_row != nullptr && old_row->GetRowId().Get() !=
3      INVALID_ROWID.Get(), "invalid old row.");
3          uint32_t                     serialized_size
4  = new_row.GetSerializedSize(schema);
5  ASSERT(serialized_size > 0, "Can not have empty row.");
6  uint32_t slot_num = old_row->GetRowId().GetSlotNum();
7  // If the slot number is invalid, abort.
8  valid = false;
```

```

9   if (slot_num >= GetTupleCount()) {
10     LOG(ERROR) << "Invalid slot number." << std::endl;
11     return false;
12   }
13   uint32_t tuple_size = GetTupleSize(slot_num);
14   // If the tuple is deleted, abort.
15   if (IsDeleted(tuple_size)) {
16     LOG(ERROR) << "Tuple already marked delete." << std::endl;
17     return false;
18   }
19   // If there is not enough space to update, we need to update
20   // via delete followed by an insert (not enough space).
21   valid = true;
22   if (GetFreeSpaceRemaining() + tuple_size < serialized_size) {
23     return false;
24   }
25 ...

```

随后在上层的 `TableHeap::UpdateTuple` 中，就可以判断更新失败的原因，如果是空间不足，就先删除再插入，可以直接调用 `TableHeap` 的接口。

还要注意这一特性会导致记录的 `RowId` 发生变化，因此上层调用时要注意维护 `RowId`。

2.2.1.2.3 Iterator

迭代器的实现比较简单，因为 `TablePage` 类提供了两个函数：`GetFirstTupleRid` 和 `GetNextTupleRid`，在遍历堆表时，只需要先查看当前行所在页是否还有下一行，如果没有，就跳到下一页获取第一行。对于 `End` 迭代器，我们可以用 `INVALID_PAGE_ID` 标识。

2.2.2 测试

我给 `TableHeap` 新增了大量测试用于保证正确性和稳定性。

- `TableHeapDeleteTest`: 测试删除后不能读取
- `TableHeapUpdateTest`: 测试原地更新和删除后插入
- `TableHeapMassiveTest`: 在该测试中，我通过严格计算插入数据的序列化长度，断言数据行应该位于的页，从而验证系统按预期运行
- `TableHeapIteratorTest`: 测试迭代器的正确性
- `HeapPerformanceTest`: 在该测试中，我控制了缓冲区大小和插入数据的总量，保证插入的数据量溢出缓冲区，之后再进行一系列更新和删除操作并验证数据，以确保模块 1 和 2 协同工作正常。事实上，`BufferPoolManager` 的一个 bug 正是在该测试中发现的。

测试结果如下：

The image shows three separate terminal windows side-by-side, each displaying the output of a different test suite. The first window on the left shows the results of the `tuple_test`, the middle window shows the results of the `table_heap_test`, and the rightmost window shows the results of the `heap_performance_test`. Each window displays a series of test cases with their status (RUN, OK, PASSED) and execution time. The `tuple_test` window has 2 tests from `TupleTest`. The `table_heap_test` window has 5 tests from `TableHeapTest`, including sample and delete tests. The `heap_performance_test` window has 1 test from `TableHeapTest`, which involves inserting 1000 tuples and performing various operations like update and massive insert.

Figure 10: `tuple_test`、`table_heap_test` 以及 `heap_performance_test` 的测试结果

2.3 Index Manager

2.3.1 原理和代码实现

第三部分主要是实现一个管理索引的 B+树，具体原理在《数据库系统》与《高级数据结构与算法》课程中都有详细介绍与举例，所以这里不详细展开，只是针对实现进行一定的介绍。

实验文档与我的代码完成过程都是自下而上的，先完成内部节点与叶子节点的设计，再完成 B+ 树的查询、插入、删除操作。但如果自下而上讲解，很难讲清楚设计思路，也难免会出现一些冗余。所以这一部分，我从 B+ 树的总体设计出发，自上而下讲解原理。

2.3.1.1 整体构建

第三部分的 B+ 树是为了加快索引查询速度而构建的，而一般会有多个索引同时存在，所以我们需要设计一定的管理机制。

具体在代码中，我选择在构建一棵新的 B+ 树的时刻（即实际插入第一条记录时），申请根节点的资源，并在 `INDEX_ROOTS_PAGE` 中进行“注册”，维护 `(Index_id, root_page_id)` 这一键值对。这样，我们可以通过 `Index_id` 区分/读取/修改/创建不同的索引。

另外，在构建 `BPlusTree` 类的对象时（在构造函数中）需要计算内部节点与叶子节点的最大容量。

这一逻辑主要体现在 `BPlusTree::BPlusTree()` 与 `BPlusTree::StartNewTree()` 中，如下面 2 段伪代码所示：

src/index/b_plus_tree.cpp

```
1 BPlusTree::Constructor(index_id,           buffer_pool_manager,
2   key_manager, leaf_max_size, internal_max_size)
3   Initialize basic tree properties
4   Set root_page_id to invalid
5
6   Attempt to load root_page_id from persistent storage
7   If fails, keep as invalid
8
9   Calculate maximum node capacities if not specified:
10    For leaf nodes based on key+rowid size
11    For internal nodes based on key+pageptr size
```

src/index/b_plus_tree.cpp

```
1 BPlusTree::StartNewTree(key, value)
2   Allocate new root page
3   If allocation fails, throw memory error
4
5   Initialize root as new leaf node:
6     Set basic properties
7     Set initial key-value pair
8     Set initial size to 1
9
10  Finalize root page setup:
11    Set next page to invalid
12    Unpin modified page
13    Update root page ID in metadata
```

由于更新 `root_page` 在各操作中是较为常见的，所以在代码中我们通过 `BPlusTree::UpdateRootPageId()` 函数封装这一操作，自动的实现注册、修改与删除索引的信息。

2.3.1.2 查询操作

B+树查询操作的接口为 `BPlusTree::GetValue()`。这一方法的实现逻辑较简单，先找到存有目标记录的叶子节点，再从叶子节点中获取记录信息即可。过程中需要使用 `InternalPage::Lookup()`，该方法通过二分查找，返回第一个大于 `key` 值的键值对的前一组键值对的 `value`。

根据 Key 值从某一节点（通常是根节点）出发查找对应叶子节点这一操作很常用，所以我们将其单独在 `BPlusTree::FindLeafPage()` 中实现。同时，考虑到 `BPlusTree::Begin()` 找到最左侧的记录与这一函数的逻辑类似，所以为该方法增加一个默认值为 `false` 的 `leftMost` 参数。我使用循环方法实现这一方法，其伪代码如下：

`src/index/b_plus_tree.cpp`

```

1 BPlusTree::FindLeafPage(key, page_id, leftMost)
2     current_page_id ← page_id
3
4     while true do
5         current_page ← fetch current_page_id
6         current_node ← cast to BPlusTreePage
7
8         if current_node is internal page then
9             if leftMost then
10                 next_page_id ← first child pointer
11             else
12                 next_page_id ← lookup key in internal node
13
14             unpin current_page
15             current_page_id ← next_page_id
16         else
17             return current_page // found leaf

```

2.3.1.3 插入操作

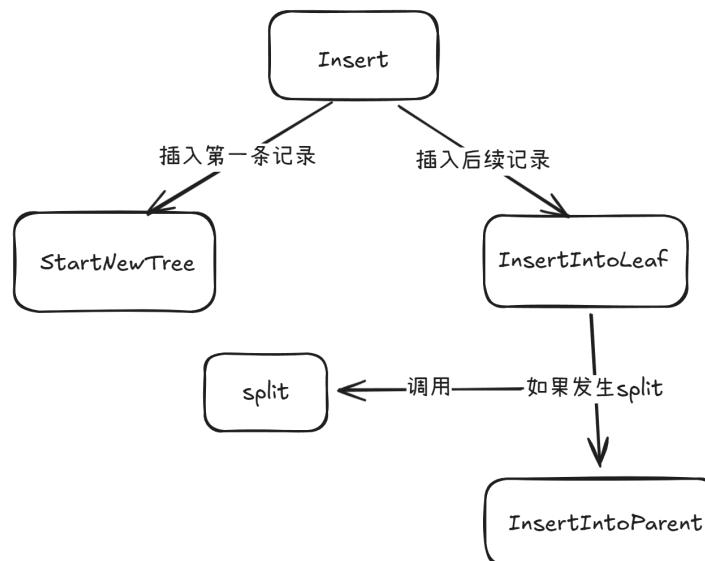


Figure 11: 插入操作主要函数关系

插入操作中相关函数的主要逻辑关系如上图所示。简单来说，插入一条记录，需要先找到对应的叶子节点，检查是否会发生分裂，若发生分裂再自下而上依次传递。

相对比较复杂的是自下而上逐层考虑分裂的 `BPlusTree::InsertIntoParent()`，这个函数我通过递归方法实现，终止情况为当前节点是合法的或当前节点是根节点。其伪代码如下：

src/index/b_plus_tree.cpp

```
1 BPlusTree::InsertIntoParent(old_node, key, new_node)
2     parent_id ← old_node.parent_page_id
3
4     if old_node is root then
5         // Case 1: Create new root
6         new_root_page ← allocate new page
7         new_root ← initialize as internal node
8         new_root.insert(old_node, key, new_node)
9             update_parent_pointers(old_node, new_node,
10                new_root.page_id)
11            update_root_metadata(new_root.page_id)
12            return
13
14        // Case 2: Insert into existing parent
15        parent_node ← fetch parent_page(parent_id)
16
17        if parent_node has space then
18            parent_node.insert(key, new_node)
19            new_node.set_parent(parent_id)
20        else
21            // Split parent
22            new_parent_node ← split(parent_node)
23
24            // Insert into appropriate split half
25            if old_node in new_parent_node then
26                new_parent_node.insert(key, new_node)
27                new_node.set_parent(new_parent_node.page_id)
28            else
29                parent_node.insert(key, new_node)
30                new_node.set_parent(parent_id)
31
32            // Recursively handle parent's parent
33            InsertIntoParent(parent_node,
34                new_parent_node.first_key, new_parent_node)
35
36        unpin_parent_pages()
```

在框架中，内部节点和叶子节点为插入操作各个函数的实现提供了充足的接口，在明确需求后实现也较为简单，这里不多赘述。

2.3.1.4 删除操作

删除操作是三大操作中逻辑最复杂的，在同一层需要考虑与相邻节点的挪借与合并，跨层次还需要考虑删除、挪借、合并带来的连锁反应。同时，因为合并操作的存在，有的节点可能在调用其他函数时被删除，所以无法完全遵循“谁申请谁销毁”的原则，于是需要通过返回值传递节点销毁信息，从而妥善管理内存。主要函数之间的关系如下图所示。

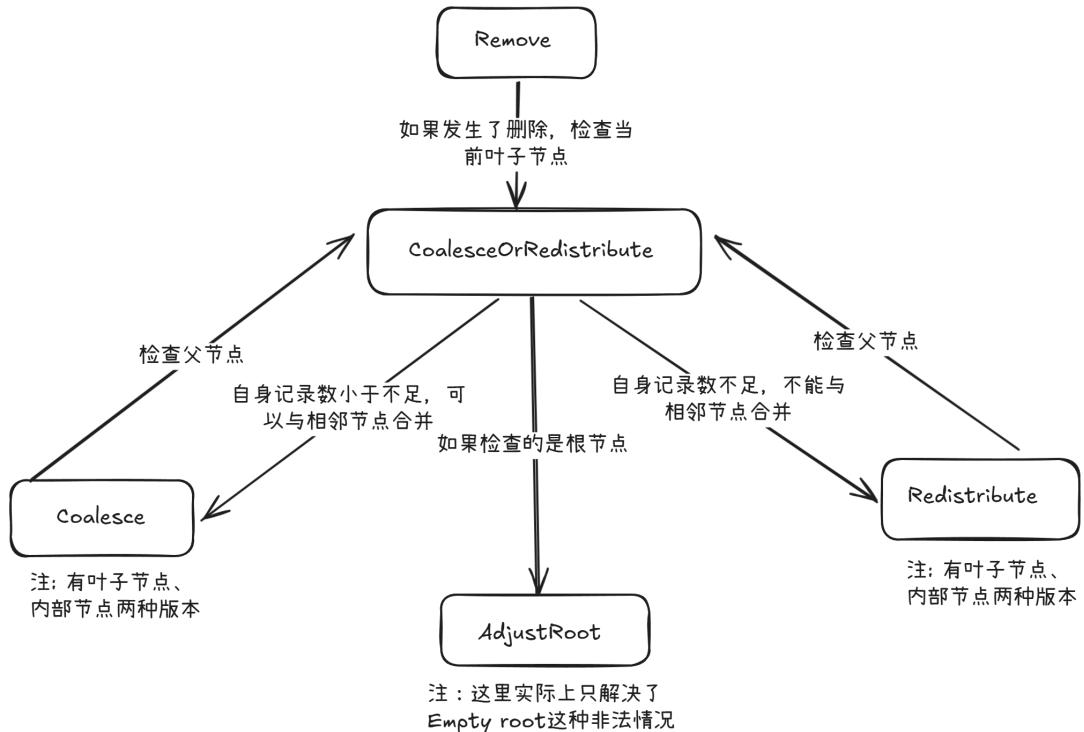


Figure 12: 删除操作主要函数关系

B+树对外的删除接口 `BPlusTree::Remove()` 的实现很简单，先找到叶子节点，删掉对应记录，最后将叶子节点传给 `BPlusTree::CoalesceOrRedistribute()` 检查是否合法。容易忽略的是，在删除叶子节点中第一个键值对后，需要逐层向上检查是否需要更新索引。

处理各种边缘情况的 `BPlusTree::CoalesceOrRedistribute()` 的实现是 B+ 树所有方法中最为复杂的。但这一方法的基本想法很简单，如果传入的节点中存储的键值对的数量过少，则通过父节点获取兄弟节点。如果可以合并成一个节点，则合并；否则传入的节点向兄弟节点挪借一个记录。为了避免类似 `Remove()` 中删除第一个记录后逐层向上检查，设置 `sibling_index = (node_index == 0) ? 1 : node_index - 1`。我使用递归方法实现，终止条件是节点合法或者节点是根节点。对于根节点的非法情况，如果根节点被删到只有一个子节点，则这个根节点直接会被删除，唯一的子节点晋升为根节点，这一过程没有在 `BPlusTree::AdjustRoot()` 中实现，因为涉及子节点的调整；另外，如果根节点

也是叶子节点，且删掉了最后一条记录，则这棵树所有资源都需要释放（包括在 INDEX_ROOTS_PAGE 中，但如果再次插入，又会再次申请）。这一方法的伪代码如下：

src/index/b_plus_tree.cpp

```
1 BPlusTree::CoalesceOrRedistribute(node)
2     if node.size ≥ minimum_size then
3         return false // No action needed
4
5     if node is root then
6         return AdjustRoot(node) // Handle root case
7
8     parent ← fetch parent of node
9     node_index ← find node's position in parent
10    sibling_index ← choose left or right sibling by node_index
11    sibling ← fetch sibling node
12
13    if nodes can be merged then
14        parent_deleted ← Coalesce(sibling, node, parent)
15
16        if parent not deleted and parent is rootpage with only
17        one child then
18            HandleRootWithSingleChild(parent)
19
20        return whether node was deleted
21    else
22        Redistribute(sibling, node) // Balance nodes
23        return false
```

Coalesce()与Redistribute()的实现我遵循的都是将左侧节点作为提供方，右侧节点作为接收方，因此需要根据 node_index 是否为 0 进行调整。

为了便于管理内存，CoalesceOrRedistribute()与Coalesce()的返回值都被用来标记传入的节点是否在过程中被删除。

在框架中，内部节点和叶子节点为删除操作各个函数的实现也提供了充足的接口，同样的，明确需求后实现并不复杂，这里不多赘述。

2.3.1.5 index_iterator

迭代器的实现比较常规，稍复杂一些的只有递增操作的实现，需要考虑跨页与重点的情况。其代码如下所示：

src/index/index_iterator.cpp

```

1 IndexIterator &IndexIterator::operator++() {
2     // Not the last item in current page
3     if (++item_index < page->GetSize()) {
4         return *this;
5     }
6
7     // Move to next page
8     item_index = 0;
9     page_id_t next_page_id = page->GetNextPageId();
10    buffer_pool_manager->UnpinPage(page->GetPageId(), false);
11
12    current_page_id = next_page_id;
13    page = (current_page_id == INVALID_PAGE_ID)
14        ? nullptr
15        : reinterpret_cast<LeafPage *>(buffer_pool_manager-
16      >FetchPage(current_page_id)->GetData());
17
18    return *this;
19 }
```

另外，`BPlusTree::End()`的注释要求与测试要求不一致，注释中要求返回最右侧纪录，测试中要求返回空值（和一般的迭代器一致）。这里遵循测试时的要求。其代码如下所示：

`src/index/index_iterator.cpp`

```

1 IndexIterator BPlusTree::End() {
2     if (IsEmpty()) return IndexIterator();
3     return IndexIterator(INVALID_PAGE_ID, buffer_pool_manager_,
4     0);
```

2.3.2 测试

由于原先 `BPlusTreeTests.SampleTest` 中的测试过于简单，最后的 B+ 树只有两层，所以我作了以下调整：

- 增大 Key 的大小

`test/index/b_plus_tree_test.cpp`

```

1 std::vector<Column *> columns = {
2     new Column("int1", TypeId::kTypeInt, 0, false, false),
```

```

3     new Column("int2", TypeId::kTypeInt, 1, false, false),
4     new Column("name", TypeId::kTypeChar, 64, 2, true, false)
5 };
6 Schema *table_schema = new Schema(columns);
7 KeyManager KP(table_schema, 128);

```

- 记录数改为 16000 条
- 在原先的末尾增加将树删空的测试

test/index/b_plus_tree_test.cpp

```

1 for (int i = n/2; i < n; i++) {
2     tree.Remove(delete_seq[i]);
3 }
4 tree.PrintTree(mgr[2], table_schema);
5 ASSERT_TRUE(tree.IsEmpty());

```

结果如图所示：

```

/mnt/d/nCoding/cpp/2025-04/minysql/build | on au :3 *1 +1
> ./test/b_plus_tree_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from BPlusTreeTests
[ RUN   ] BPlusTreeTests.SampleTest
[ OK    ] BPlusTreeTests.SampleTest (7572 ms)
[-----] 1 test from BPlusTreeTests (7572 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (7572 ms total)
[ PASSED ] 1 test.

/mnt/d/nCoding/cpp/2025-04/minysql/build | on au :3 *1 +1
> ./test/b_plus_tree_index_test
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from BPlusTreeTests
[ RUN   ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[ OK    ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (0 ms)
[ RUN   ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[ OK    ] BPlusTreeTests.BPlusTreeIndexSimpleTest (121 ms)
[-----] 2 tests from BPlusTreeTests (122 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (122 ms total)
[ PASSED ] 2 tests.

/mnt/d/nCoding/cpp/2025-04/minysql/build | on au :3 *1 +1
> ./test/index_iterator_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from BPlusTreeTests
[ RUN   ] BPlusTreeTests.IndexIteratorTest
[ OK    ] BPlusTreeTests.IndexIteratorTest (130 ms)
[-----] 1 test from BPlusTreeTests (130 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (130 ms total)
[ PASSED ] 1 test.

```

Figure 13: #3 测试结果

2.4 Catalog Manager

2.4.1 原理和代码实现

Catalog Manager（目录管理器）负责管理数据库的所有模式信息（包括数据库内所有的表，及其字段和索引的定义信息）以及访问这些信息的接口，供下一个模块 Executor 使用。不难发现，该模块和其余几个模块都有一定的联系，足以体现其重要性。

2.4.1.1 目录元信息

首先，我们要补充目录元数据、表元数据和索引元数据的获取序列化结果长度的方法 `GetSerializedSize` 函数。这类方法会在实现序列化的方法 `SerializeTo` 的开头被调用，以预先判断页的大小是否能容纳序列化的结果，如果可以的话就继续序列化，否则宣告失败。

我们可以根据 `SerializeTo` 中 `buf` 指针的移动步幅来推出序列化结果的大小，因此实现比较简单。

src/catalog/catalog.cpp

```
1 uint32_t CatalogMeta::GetSerializedSize() const {
2     uint32_t serialized_size = 12;
3     uint32_t table_meta_pages_size = table_meta_pages_.size();
4     uint32_t index_meta_pages_size = index_meta_pages_.size();
5
6         serialized_size += 8 * (table_meta_pages_size +
7 index_meta_pages_size);
8
9     return serialized_size;
}
```

src/catalog/table.cpp

```
1 uint32_t TableMetadata::GetSerializedSize() const {
2     return 4 + 4 + MACH_STR_SERIALIZED_SIZE(table_name_) + 4 +
3 schema_→GetSerializedSize();
```

src/catalog/indexes.cpp

```
1 uint32_t IndexMetadata::GetSerializedSize() const {
```

```

2         uint32_t      serialized_size      =      16      +
3 MACH_STR_SERIALIZED_SIZE(index_name_);
4     uint32_t key_map_size = key_map_.size();
5     serialized_size += 4 * key_map_size;
6
7 } 
```

接下来完善 IndexInfo 类的 Init 方法。我们可以根据代码框架中提供的注释一步步实现代码：

src/include/catalog/indexes.h

```

1 void  Init(IndexMetadata  *meta_data,  TableInfo  *table_info,
2 BufferPoolManager *buffer_pool_manager) {
3     // Step1: init index metadata and table info
4     // Step2: mapping index key to key schema
5     // Step3: call CreateIndex to create the index
6     this->meta_data_ = meta_data;
7     key_schema_ = Schema::ShallowCopySchema(table_info-
8 >GetSchema(), meta_data_->GetKeyMapping());
9     index_ = CreateIndex(buffer_pool_manager, "bptree");
10 } 
```

由于注释写的很清楚了，这里就不再解释一遍了。不过需要注意的是，索引模式 key_schema_ 是需要浅拷贝的。

2.4.1.2 表和索引的管理

本模块的重头戏在于利用目录、表和索引的元数据，实现对表和索引的管理。我们需要完成以下方法的实现：

- CatalogManager::CatalogManager：构造函数

src/catalog/catalog.cpp

```

1 CatalogManager::CatalogManager(BufferPoolManager
2 *buffer_pool_manager, LockManager *lock_manager,
3                                     LogManager *log_manager,
4     bool init)
5             : buffer_pool_manager_(buffer_pool_manager),
6             lock_manager_(lock_manager), log_manager_(log_manager) {
7     table_names_.clear(); 
```

```

5   tables_.clear();
6   index_names_.clear();
7   indexes_.clear();
8
9   if (init) {
10     // initialize(create) the catalog meta
11     catalog_meta_ = CatalogMeta::NewInstance();
12     // catalog_meta_->SerializeTo(buf);
13   } else {
14     // fetch catalog meta page from the disk
15     Page* page = buffer_pool_manager-
16     >FetchPage(CATALOG_META_PAGE_ID);
17     char* buf = page->GetData();
18     // load the catalog meta from the buffer pool
19     catalog_meta_ = catalog_meta_->DeserializeFrom(buf);
20     // load table meta pages
21     for (const auto& table_meta_page: catalog_meta_-
22       >table_meta_pages_) {
23       LoadTable(table_meta_page.first,
24       table_meta_page.second);
25     }
26     // load index meta pages
27     for (const auto& index_meta_page: catalog_meta_-
28       >index_meta_pages_) {
29       LoadIndex(index_meta_page.first,
30       index_meta_page.second);
31     }
32     buffer_pool_manager->UnpinPage(CATALOG_META_PAGE_ID,
33     false);
34   }
35 }
```

- 先清空 CatalogManager 中关于表和索引相关属性的内容。
- 若 `init = true`, 即需要初始化, 则从头创建一个新的 `catalog_meta` 实例。
- 否则, 就要从磁盘中加载元数据, 具体步骤为:
 - 从磁盘中取出包含目录元数据的页, 放入缓存池中, 然后将数据反序列化, 得到目录元数据。
 - 根据目录元数据的信息, 我们也就知道了表和目录的元数据, 包括它们的名称和 id。所以就通过后面要实现的 `LoadTable` 和 `LoadIndex` 方法, 以迭代方式加载所有的表和索引的元数据。
 - 最后要将目录元数据对应的数据页解除固定
- `CatalogManager::CreateTable`: 创建表

src/catalog/catalog.cpp

```
1 dberr_t      CatalogManager::CreateTable(const      string
2   &table_name,  TableSchema  *schema,    Txn   *txn,    TableInfo
3   *&table_info) {
4     try {
5       // check if the table with same name exists
6       if (table_names_.count(table_name) == 1)
7         return DB_TABLE_ALREADY_EXIST;
8
9       table_id_t    table_id    =    catalog_meta_-
10      >GetNextTableId();    // get new table id
11      TableMetadata *table_meta = nullptr;
12      TableHeap    *table_heap = nullptr;
13      TableSchema   *table_schema = nullptr;
14      page_id_t    table_page_id, page_id;
15      Page        *table_page = nullptr, *table_meta_page = nullptr,
16      *catalog_meta_page = nullptr;
17      char        *buf;
18
19      // get new pages from buffer pool for table data
20      //table_page    =    buffer_pool_manager_-
21      >NewPage(table_page_id);
22      table_meta_page    =    buffer_pool_manager_-
23      >NewPage(page_id);
24
25      // deep copy from passed schema
26      table_schema = table_schema->DeepCopySchema(schema);
27
28      // create and initialize new table info
29      table_heap    =    table_heap->Create(buffer_pool_manager_,
30      table_schema, txn, log_manager_, lock_manager_);
31      table_page_id = table_heap->GetFirstPageId();
32      table_meta    =    table_meta->Create(table_id, table_name,
33      table_page_id, table_schema);
34      table_meta->SerializeTo(table_meta_page->GetData());
35      buffer_pool_manager_->UnpinPage(page_id, true);
36      table_info    =    table_info->Create();
37      table_info->Init(table_meta, table_heap);
38
39      // update catalog manager
40      table_names_[table_name] = table_id;
41      tables_[table_id] = table_info;
42
43      // update catalog metadata
44      catalog_meta_->table_meta_pages_[table_id] = page_id;
```

```

    catalog_meta_page = buffer_pool_manager_-
37 >FetchPage(CATALOG_META_PAGE_ID);
38 buf = catalog_meta_page->GetData();
39 catalog_meta_->SerializeTo(buf);
40 buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID,
41 true);
42 return DB_SUCCESS;
43 } catch (exception e) {
44     return DB_FAILED;
45 }
46 }
```

- 首先检查数据库中是否存在和指定表名相同的表，若存在则返回 DB_TABLE_ALREADY_EXIST 错误码，并终止调用。
- 然后要构造表信息（包括了表的元数据和堆表数据结构）。需要注意的点有：
 - 表模式需要深拷贝。
 - 因为表的元数据需要持久化存储，因此将其对应的数据页标记为脏页。
 - 接着用已知的表明、表 id 和表信息更新目录管理器的内容。
 - 最后要更新目录元数据，记得标记为脏页。
- CatalogManager::GetTable: 获取表信息

src/catalog/catalog.cpp

```

1 dberr_t CatalogManager::GetTable(const string &table_name,
2 TableInfo *&table_info) {
3     // check if the specified table exists
4     if (table_names_.count(table_name) == 0)
5         return DB_TABLE_NOT_EXIST;
6
7     // get table info from catalog manager
8     table_id_t table_id = table_names_[table_name];
9     table_info = tables_[table_id];
10
11 }
```

- 如果指定表名的表不存在，返回错误码 DB_TABLE_NOT_EXIST，并终止程序。
- 否则从目录管理器中获取对应的表信息。
- CatalogManager::GetTables: 获取全部表信息

src/catalog/catalog.cpp

```
1 dberr_t CatalogManager::GetTables(vector<TableInfo * >
2 &tables) const {
3     // ensure that variable tables has enough space for all
4     // tables in db
5     tables.resize(tables_.size());
6     tables.clear();
7     // get all table info
8     for (const auto& table : tables_)
9         tables.push_back(table.second);
10 }
```

- 读取全部表信息前，需要调整一下 `tables` 的大小，确保其能容纳全部表信息。
- `CatalogManager::CreateIndex`: 创建索引

src/catalog/catalog.cpp

```
1 dberr_t CatalogManager::CreateIndex(const std::string
2 &table_name, const string &index_name,
3                                     const std::vector<std::string>
4 &index_keys, Txn *txn, IndexInfo *&index_info,
5                                     const string &index_type) {
6     try {
7         // check if the specified table exists
8         if (table_names_.count(table_name) == 0)
9             return DB_TABLE_NOT_EXIST;
10
11        // check if the index with same name exists
12        if (index_names_.count(table_name) == 1) {
13            auto index_list = index_names_[table_name];
14            if (index_list.count(index_name) == 1)
15                return DB_INDEX_ALREADY_EXIST;
16        }
17
18        table_id_t table_id = table_names_[table_name];
19        TableInfo *table_info = tables_[table_id];
20        TableMetadata *table_meta = nullptr;
21        TableHeap *table_heap = table_info->GetTableHeap();
22        TableSchema *table_schema = table_info->GetSchema();
```

```

22         std::unordered_map<std::string,    index_id_t>
23     index_name_to_index;
24     std::vector<uint32_t> key_map;
25     index_id_t index_id = catalog_meta->GetNextIndexId();
26     IndexMetadata *index_meta = nullptr;
27     IndexSchema *key_schema = nullptr;
28
29     page_id_t index_page_id, page_id;
30     Page *index_page, *index_meta_page, *catalog_meta_page;
31     char *buf;
32
33     // get new pages from buffer pool for index data
34     //      index_page      =      buffer_pool_manager-
35     >NewPage(index_page_id);
36     //      index_meta_page      =      buffer_pool_manager-
37     >NewPage(page_id);
38
39     // obtain key_map
40     uint32_t col_index;
41     for (const auto& col_name : index_keys) {
42         if (table_schema->GetColumnIndex(col_name, col_index)
43 == DB_COLUMN_NAME_NOT_EXIST)
44             return DB_COLUMN_NAME_NOT_EXIST;
45         key_map.push_back(col_index);
46     }
47
48     // create and initialize index info
49     index_meta = index_meta->Create(index_id, index_name,
50     table_id, key_map);
51     index_meta->SerializeTo(index_meta_page->GetData());
52     buffer_pool_manager_->UnpinPage(page_id, true);
53     index_info = index_info->Create();
54         index_info->Init(index_meta,    table_info,
55     buffer_pool_manager_);
56
57     // Get the table iterator for all records in the table
58     key_schema = index_info->GetIndexKeySchema();
59     for (TableIterator it = table_heap->Begin(nullptr); it !
= table_heap->End(); ++it) {
56         // Get the current row and insert its key into
57         the index
58         Row row = *it;
59         Row key_row;
60         row.GetKeyFromRow(table_schema, key_schema, key_row);
61         index_info->GetIndex()->InsertEntry(key_row,
62         row.GetRowId(), nullptr);
63     }

```

```

60
61     // update catalog manager
62     // If the table does not have any indexes yet, create a
63     new map for it
64     if (index_names_.find(table_name) == index_names_.end())
65     {
66         index_names_[table_name] = std::unordered_map<std::string, index_id_t>();
67     }
68     index_names_[table_name][index_name] = index_id;
69     indexes_[index_id] = index_info;
70
71     // update catalog metadata
72     catalog_meta_>index_meta_pages_[index_id] = page_id;
73     catalog_meta_page = buffer_pool_manager_
74     >FetchPage(CATALOG_META_PAGE_ID);
75     buf = catalog_meta_page->GetData();
76     catalog_meta_->SerializeTo(buf);
77     buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID,
78     true);
79
80     return DB_SUCCESS;
81 } catch (exception e) {
82     return DB_FAILED;
83 }
84 }
```

- 先做两步检查：
 - 确保指定表名的表存在；
 - 确保指定索引名的索引不存在。
- 如果没问题的话就继续构建索引信息。其步骤和前面构造表信息的过程十分类似，除了实现细节上有些差别。
- 最后更新目录元数据的步骤几乎和 CreateTable 的一样，故不再赘述。
- CatalogManager::GetIndex: 获取索引信息

src/catalog/catalog.cpp

```

1 dberr_t CatalogManager::GetIndex(const std::string
2   &table_name, const std::string &index_name,
3                                     IndexInfo *&index_info)
4
5 const {
6     // check if the specified table exists
7     if (index_names_.count(table_name) == 0)
```

```

5     return DB_TABLE_NOT_EXIST;
6
7     // check if the specified index exists
8     auto index_list = index_names_.at(table_name);
9     if (index_list.count(index_name) == 0)
10        return DB_INDEX_NOT_FOUND;
11
12    // get index info from catalog manager
13    index_id_t index_id
14    = index_names_.at(table_name).at(index_name);
15    index_info = indexes_.at(index_id);
16
17    return DB_SUCCESS;
18 }
```

- 同样要先检查指定表和索引是否存在，否则返回相应的错误码并终止程序。
- 若通过检查，则读取指定表下指定索引的信息。
- 注意该方法声明为 `const`，无法用方括号访问私有属性，需要用 `.at()` 方法实现安全访问。
- `CatalogManager::GetTableIndexes`: 获取指定表的全部索引信息

src/catalog/catalog.cpp

```

1  dberr_t CatalogManager::GetTableIndexes(const std::string
2   &table_name, std::vector<IndexInfo *> &indexes) const {
3     // check if the specified table exists
4     if (index_names_.count(table_name) == 0)
5       return DB_TABLE_NOT_EXIST;
6
7     // ensure that variable indexes has enough space for all
8     // indexes in the table
9     indexes.resize(indexes_.size());
10    indexes.clear();
11    // get all index info in the table
12    auto index_list = index_names_.at(table_name);
13    for (const auto& index : index_list)
14      indexes.push_back(indexes_.at(index.second));
15
16    return DB_SUCCESS;
17 }
```

- 先检查指定表是否存在，若不存在则返回错误码 `DB_TABLE_NOT_EXIST`，并终止程序。

- 否则读取全部索引信息。在读取之前同样调整一下 `indexes` 的大小，确保其能容纳全部索引信息。
- `CatalogManager :: DropTable`: 删除表

src/catalog/catalog.cpp

```

1 dberr_t CatalogManager::DropTable(const string &table_name)
2 {
3     // cannot delete non-existent table
4     if (table_names_.count(table_name) == 0)
5         return DB_TABLE_NOT_EXIST;
6
7     // delete the entry from table_names_
8     table_id_t table_id = table_names_[table_name];
9     table_names_.erase(table_name);
10
11    // exception
12    if (tables_.count(table_id) == 0)
13        return DB_FAILED;
14
15    // delete and deallocate table info
16    TableInfo* table = tables_[table_id];
17    tables_.erase(table_id);
18    delete table;
19
20    return DB_SUCCESS;
21 }
```

- 先检查指定表是否存在，若不存在则返回错误码 `DB_TABLE_NOT_EXIST`，并终止程序。
- 清空目录管理器中有关该表的信息，包括其 id 和名称。
- 最后释放表信息（包括了元数据和堆表）的存储空间。
- `CatalogManager :: DropIndex`: 删除索引

src/catalog/catalog.cpp

```

1 dberr_t CatalogManager::DropIndex(const string &table_name,
2                                   const string &index_name) {
3     // cannot delete non-existent table and non-existent index
4     if (table_names_.count(table_name) == 0)
5         return DB_TABLE_NOT_EXIST;
6
7     if (index_names_.count(table_name) == 1) {
8         auto index_list = index_names_.at(table_name);
```

```

8     if (index_list.count(index_name) == 0)
9         return DB_INDEX_NOT_FOUND;
10    } else {
11        return DB_FAILED;
12    }
13
14    // delete the entry from index_names_
15    index_id_t index_id = index_names_[table_name]
16    [index_name];
17    index_names_[table_name].erase(index_name);
18
19    // exception
20    if (indexes_.count(index_id) == 0)
21        return DB_FAILED;
22
23    // delete and deallocate index info
24    IndexInfo *index_info = indexes_[index_id];
25    indexes_.erase(index_id);
26    delete index_info;
27
28    return DB_SUCCESS;
29 }
```

- 先检查指定表和索引是否存在，若不存在则返回相应的错误码并终止程序。
- 清空目录管理器中有关该表索引的信息，包括其 id 和名称等。
- 最后释放索引信息的存储空间。
- CatalogManager::FlushCatalogMetaPage: 将目录元数据转储到磁盘中

src/catalog/catalog.cpp

```

1 dberr_t CatalogManager::FlushCatalogMetaPage() const {
2     // do as the method name indicates
3     if (buffer_pool_manager_>FlushPage(CATALOG_META_PAGE_ID))
4         return DB_SUCCESS;
5     return DB_FAILED;
6 }
```

- CatalogManager::LoadTable: 加载表信息

src/catalog/catalog.cpp

```

1 dberr_t CatalogManager::LoadTable(const table_id_t
2     table_id, const page_id_t page_id) {
```

```

2   try {
3     // cannot load non-existent table
4     if (tables_.count(table_id) ≠ 0)
5       return DB_TABLE_ALREADY_EXIST;
6
7     TableMetadata *table_meta = nullptr;
8     TableInfo *table_info = nullptr;
9     TableHeap *table_heap = nullptr;
10    TableSchema *table_schema = nullptr;
11    page_id_t table_page_id;
12    Page *page;
13    char *buf;
14
15    // load page from disk
16    page = buffer_pool_manager_→FetchPage(page_id);
17    buf = page→GetData();
18
19    // obtain table metadata
20    table_meta→DeserializeFrom(buf, table_meta);
21
22    // create table info
23    table_page_id = table_meta→GetFirstPageId();
24    table_schema = table_meta→GetSchema();
25    table_heap = table_heap→Create(buffer_pool_manager_,
26        table_page_id, table_schema, log_manager_, lock_manager_);
27    table_info = table_info→Create();
28    table_info→Init(table_meta, table_heap);
29
30    // update catalog manager
31    std::string table_name = table_meta→GetTableName();
32    table_names_[table_name] = table_id;
33    tables_[table_id] = table_info;
34
35  } catch (exception e) {
36    return DB_FAILED;
37  }
38 }

```

- 如果目录管理器已经有对应的表了，就不能重复加载。
- 否则就继续加载表：
 - 先从磁盘中获取该表对应的数据页；
 - 反序列化该数据页，得到表的元数据；
 - 根据元数据创建堆表，并形成表信息；
 - 更新目录管理器相关内容。
- CatalogManager :: LoadIndex：加载索引信息

```
src/catalog/catalog.cpp
```

```
1 dberr_t CatalogManager::LoadIndex(const index_id_t
2 index_id, const page_id_t page_id) {
3     try {
4         // cannot load non-existent index
5         if (indexes_.count(index_id) != 0)
6             return DB_INDEX_ALREADY_EXIST;
7
8         table_id_t table_id;
9         TableMetadata *table_meta = nullptr;
10        TableInfo *table_info = nullptr;
11
12        IndexMetadata *index_meta = nullptr;
13        IndexInfo *index_info = nullptr;
14
15        page_id_t index_page_id, table_page_id;
16        Page *page;
17        char *buf;
18
19        // load page from disk
20        page = buffer_pool_manager_→FetchPage(page_id);
21        buf = page→GetData();
22
23        // obtain index metadata
24        index_meta→DeserializeFrom(buf, index_meta);
25
26        // create index info
27        table_id = index_meta→GetTableId();
28        table_info = tables_[table_id];
29        index_info = index_info→Create();
30        index_info→Init(index_meta, table_info,
31                         buffer_pool_manager_);
32
33        // update catalog manager
34        std::string index_name = index_meta→GetIndexName();
35        std::string table_name = table_info→GetTableName();
36        index_names_[table_name][index_name] = index_id;
37        indexes_[index_id] = index_info;
38
39        return DB_SUCCESS;
40    } catch (exception e) {
41        return DB_FAILED;
42    }
43 }
```

- 思路上和 LoadTable 是一致的，只是实现细节的不同，这里就不展开解释了。

- CatalogManager::GetTable: 直接根据表 id 获取表信息（重载）

```
src/catalog/catalog.cpp
```

```

1 dberr_t CatalogManager::GetTable(const table_id_t table_id,
2     TableInfo *&table_info) {
3     if (tables_.count(table_id) == 0)
4         return DB_TABLE_NOT_EXIST;
5
6     // get table info directly by table id
7     table_info = tables_[table_id];
8
9     return DB_SUCCESS;
}
```

2.4.2 测试

先来看已有的测试代码：

- CatalogMetaTest: 针对 CatalogMeta 类做测试，检查其是否有保存表和索引元数据的能力。
- CatalogTableTest: 和表相关的测试，检查能否创建和读取表，并且能否在关闭数据库，打开新的数据库后成功加载表。
- CatalogIndexTest: 和索引相关的测试，检查能否创建和读取索引，并且能否在关闭数据库，打开新的数据库后成功加载索引。

测试结果如下：

```

./test/catalog_test
[=====] Running 3 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 3 tests from CatalogTest
[ RUN   ] CatalogTest.CatalogMetaTest
[ OK    ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN   ] CatalogTest.CatalogTableTest
[ OK    ] CatalogTest.CatalogTableTest (279 ms)
[ RUN   ] CatalogTest.CatalogIndexTest
[ OK    ] CatalogTest.CatalogIndexTest (280 ms)
[-----] 3 tests from CatalogTest (560 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test suite ran. (560 ms total)
[ PASSED ] 3 tests.

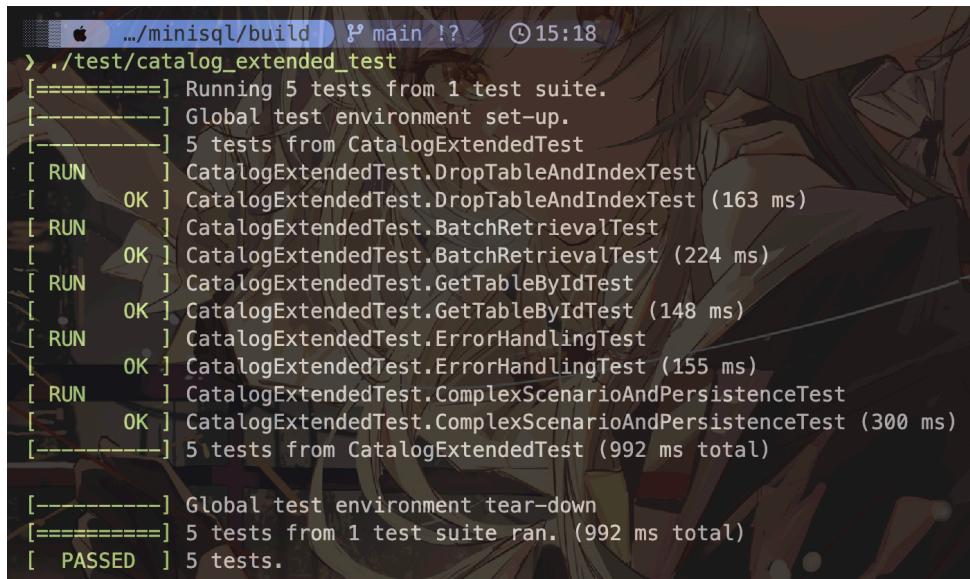
```

Figure 14: 通过 catalog_test 的测试

不过已有的测试代码较为简单，并没有覆盖到我们在该模块实现的全部代码。因此，我们自行设计了以下测试代码（具体可见 test/catalog/catalog_extended_test.cpp），以尽可能覆盖到可能出现的情况：

- `DropTableAndIndexTest`：测试表和索引的创建、删除，以及删除不存在的表和索引时的错误处理（测试更多的功能）。
- `BatchRetrievalTest`：测试批量创建表和索引，以及批量获取所有表和指定表的所有索引。
- `GetTableByIdTest`：测试通过表的 id 获取表对象，以及不存在 id 的处理（trivial case，只是为了覆盖所有实现方法）。
- `ErrorHandlingTest`：测试重复表名、重复索引名、在不存在的表上创建索引、使用不存在的列名创建索引等错误场景。
- `ComplexScenarioAndPersistenceTest`：测试复杂多表多索引场景下的创建、持久化与重载后数据一致性，以及删除操作的正确性（采用经典的大学数据库案例）。

测试结果如下：



```

./minisql/build ./main !? @ 15:18
> ./test/catalog_extended_test
[=====] Running 5 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 5 tests from CatalogExtendedTest
[ RUN   ] CatalogExtendedTest.DropTableAndIndexTest
[ OK    ] CatalogExtendedTest.DropTableAndIndexTest (163 ms)
[ RUN   ] CatalogExtendedTest.BatchRetrievalTest
[ OK    ] CatalogExtendedTest.BatchRetrievalTest (224 ms)
[ RUN   ] CatalogExtendedTest.GetTableByIdTest
[ OK    ] CatalogExtendedTest.GetTableByIdTest (148 ms)
[ RUN   ] CatalogExtendedTest.ErrorHandlingTest
[ OK    ] CatalogExtendedTest.ErrorHandlingTest (155 ms)
[ RUN   ] CatalogExtendedTest.ComplexScenarioAndPersistenceTest
[ OK    ] CatalogExtendedTest.ComplexScenarioAndPersistenceTest (300 ms)
[-----] 5 tests from CatalogExtendedTest (992 ms total)

[-----] Global test environment tear-down
[-----] 5 tests from 1 test suite ran. (992 ms total)
[ PASSED ] 5 tests.

```

Figure 15: 通过 `catalog_extended_test` 的测试

2.5 Planner and Executor

2.5.1 原理

本实验主要包括 Planner 和 Executor 两部分：

- Planner 的主要功能是将解释器(Parser)生成的语法树，改写成数据库可以理解的数据结构。在这个过程中，我们会将所有 SQL 语句中的标识符解析成没有歧义的实体，即各种 C++ 的类，并通过 Catalog Manager 提供的信息生成执行计划。
 - 其中 Parser 的原理涉及到编译原理的知识，我们还没有学过，因此代码框架中已实现了这部分的内容。我们只需理解 Parser 生成的语法树的数据结构，并知道它支持解析哪些 SQL 语句即可。

- 对于简单 SQL 语句，生成的语法树也非常简单，可能只有一两个节点。此时无需传入 Planner 生成执行计划，我们直接调用对应的执行函数执行即可。
- 而复杂 SQL 语句对应的语法树也较为复杂。生成的语法树需传入 Planner 生成执行计划，并交由 Executor 进行执行。
 - Planner 需要先遍历语法树，调用 Catalog Manager 检查语法树中的信息是否正确，如表、列是否存在，谓词的值类型是否与列类型对应等等，随后将这些词语抽象成相应的表达式，即可以理解的各种 C++ 类。
 - 解析完成后，Planner 根据改写语法树后生成的可以理解的 Statement 结构，生成对应的 Plannode，并将 Plannode 交由 Executor 进行执行。
- Executor 遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor，随后调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，并将执行结果返回给上层模块。
- 本任务采用的是最经典的 Iterator Model，即火山模型。
 - 执行引擎会将整个 SQL 语句 · 构建成一个算子树。
 - 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算子，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。
 - MiniSQL 中涉及到的算子包括：
 - **SeqScanExecutor**: 对表执行一次顺序扫描，一次 `Next()` 方法返回一个符合谓词条件的行。顺序扫描的表名和谓词由 SeqScanPlanNode 指定。
 - **IndexScanExecutor**: 对表执行一次带索引的扫描，一次 `Next()` 方法返回一个符合谓词条件的行。为简单起见，IndexScan 仅支持单列索引。当 Planner 检测到 `select` 的谓词中的列上存在索引，而且索引只包含该列时，会生成 IndexScanPlan，其他情况则生成 SeqScanPlan。
 - **InsertExecutor**: 将行插入表中并更新索引。要插入的值通过 ValueExecutor 生成对应的行，随后被拉取到 InsertExecutor 中。
 - **UpdateExecutor**: 修改指定表中的现有行并更新其索引。UpdatePlanNode 将有一个 SeqScanPlanNode 作为其子节点，要更新的值通过 SeqScanExecutor 提供。
 - **DeleteExecutor**: 删除表中符合条件的行。和 Update 一样，DeletePlanNode 将有一个 SeqScanPlanNode 作为其子节点，要删除的值通过 SeqScanExecutor 提供。

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。

- 本实验需要我们需要实现的是 `src/include/executor/execute_engine.h` 中的创建/删除/查询数据库、数据表、索引等函数。它们对应的语法树较为简单，因此不用通过 Planner 生成查询计划。这些函数包括：

- `ExecuteEngine::ExecuteCreateTable`
- `ExecuteEngine::ExecuteDropTable`
- `ExecuteEngine::ExecuteShowIndexes`
- `ExecuteEngine::ExecuteCreateIndex`
- `ExecuteEngine::Execute DropIndex`
- `ExecuteEngine::ExecuteExecfile`
- `ExecuteEngine::ExecuteQuit`

具体代码实现将在下一小节给出。

2.5.2 代码实现

为便于阅读和理解，下面为每个要实现的函数附一张语法树的结构图。

- `ExecuteEngine::ExecuteCreateTable`: 执行形如 `create table ...` 的 SQL 语句

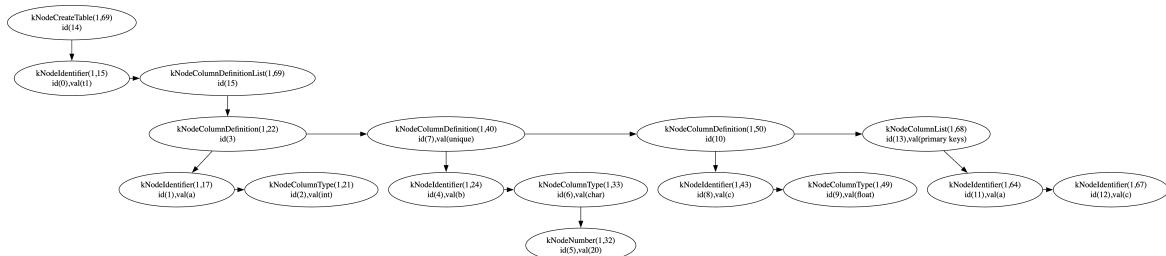


Figure 16: `ExecuteCreateTable` 要执行的语法树

`src/executor/execute_engine.cpp`

```

1 dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast,
2 ExecuteContext *context) {
3 #ifdef ENABLE_EXECUTE_DEBUG
4   LOG(INFO) << "ExecuteCreateTable" << std::endl;
5 #endif
6   if (current_db_.empty()) {
7     cout << "No database selected" << endl;
8     return DB_FAILED;
  
```

```

8 }
9
10 auto catalog = context->GetCatalog();
11 string table_name = ast->child_->val_;
12 Column *column;
13 vector<Column *> columns;
14 set<string> primary_keys;
15 auto node = ast->child_->next_->child_;
16 int table_idx = 0;
17 int char_len;
18
19 // check if there is a primary key constraint
20 while (node) {
21     if (node->type_ == kNodeColumnList) {
22         if (string(node->val_) == "primary keys") {
23             auto cur = node->child_;
24             while (cur) {
25                 primary_keys.insert(cur->val_);
26                 cur = cur->next_;
27             }
28         }
29         break;
30     }
31     node = node->next_;
32 }
33
34 // collect all columns
35 node = ast->child_->next_->child_;
36 while (node) {
37     if (node->type_ != kNodeColumnDefinition) {
38         node = node->next_;
39         continue;
40     }
41
42     string column_name = node->child_->val_;
43     string column_type = node->child_->next_->val_;
44     bool is_unique = false;
45     bool is_null = false;
46
47     // integrity constraints
48     if (node->val_ != NULL) {
49         if (string(node->val_) == "unique") {
50             is_unique = true;
51         }
52         if (string(node->val_) == "is null")
53             is_null = true;
54     } else if (primary_keys.count(column_name) != 0) {
55         is_unique = true;
56         is_null = true;

```

```

57     }
58
59     // column types
60     auto type = kTypeInvalid;
61     if (column_type == "int")
62         type = kTypeInt;
63     else if (column_type == "char") {
64         type = kTypeChar;
65         char_len = atoi(node->child_->next_->child_->val_);
66     }
67     else if (column_type == "float")
68         type = kTypeFloat;
69
70     // create a new column
71     if (type != kTypeChar) {
72         column = new Column(column_name, type, table_idx,
73         is_null, is_unique);
74     } else {
75         column = new Column(column_name, type, char_len,
76         table_idx, is_null, is_unique);
77     }
78     columns.push_back(column);
79     ++table_idx;
80
81     node = node->next_;
82 }
83
84 // create a new table
85 auto schema = new Schema(columns);
86 TableInfo *table_info;
87 auto err_msg = catalog->CreateTable(table_name, schema,
88 nullptr, table_info);
89 if (err_msg != DB_SUCCESS) {
90     return err_msg;
91 }
92
93 // create index for primary keys
94 if (!primary_keys.empty()) {
95     auto pk_index_name = "pk_" + table_name;
96     IndexInfo *pk_index_info;
97     vector<string> pk_index_keys(primary_keys.begin(),
98     primary_keys.end());
99     if (catalog->CreateIndex(table_name, pk_index_name,
100    pk_index_keys, nullptr, pk_index_info, "btree") !=
101    DB_SUCCESS) {
102         LOG(WARNING) << "Failed to create primary key index
103         for table " << table_name;
104         return DB_FAILED;

```

```

98     }
99 }
100
101    // create unique indexes for columns
102    for (const auto& col : columns) {
103        if (col->IsUnique() && primary_keys.count(col-
104 >GetName()) == 0) {
105            vector<string> uq_col_name = {col->GetName()};
106            auto uq_index_name = "uk_" + table_name + "_" +
107 col->GetName();
108            IndexInfo *uq_index_info;
109            if (catalog->CreateIndex(table_name, uq_index_name,
110 uq_col_name, nullptr, uq_index_info, "btree") !=
111 DB_SUCCESS) {
112                LOG(WARNING) << "Failed to create unique index for
113 column " << col->GetName() << " in table " << table_name;
114                return DB_FAILED;
115            }
116        }
117    }
118
119    return DB_SUCCESS;
120 }

```

- 先找出语句中出现的主键约束，如果有的话将这些列名记录在 `primary_keys` 中。
- 然后遍历每个节点，找出每个列的名称、类型、完整性约束等信息，并根据这些信息构造出一个 `Column` 对象，放进 `columns` 容器中，之后用作创建表的模式。
- 创建表。
- 为主键创建索引。
- 为每个有 `unique` 约束的列单独创建索引（因为 MiniSQL 中的 B+ 树索引默认只支持唯一键）
- `ExecuteEngine :: ExecuteDropTable`: 执行形如 `drop table ...` 的 SQL 语句

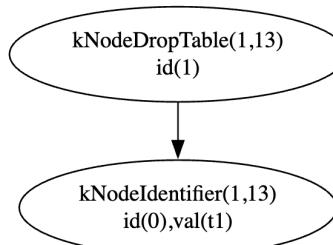


Figure 17: `ExecuteDropTable` 要执行的语法树

```
src/executor/execute_engine.cpp
```

```
1 dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast,
2 ExecuteContext *context) {
3 #ifdef ENABLE_EXECUTE_DEBUG
4     LOG(INFO) << "ExecuteDropTable" << std::endl;
5 #endif
6     if (current_db_.empty()) {
7         cout << "No database selected" << endl;
8         return DB_FAILED;
9     }
10    auto catalog = context->GetCatalog();
11    string table_name = ast->child_->val_;
12
13    // drop all indexes of the table first
14    vector<IndexInfo *> indexes;
15    catalog->GetTableIndexes(table_name, indexes);
16    for (const auto &index : indexes) {
17        auto index_name = index->GetIndexName();
18        auto err_msg = catalog-> DropIndex(table_name,
19 index_name);
20        if (err_msg != DB_SUCCESS) {
21            cout << "Failed to drop index " << index_name << endl;
22            return err_msg;
23        }
24    }
25    // then drop the table
26    auto err_msg = catalog->DropTable(table_name);
27    if (err_msg != DB_SUCCESS) {
28        cout << "Failed to drop table " << table_name << endl;
29        return err_msg;
30    }
31
32    return DB_SUCCESS;
33 }
```

- 首先要删除该表下的所有索引。如果先把表删掉的话，那么那些索引就没法访问了，而且还占据着存储空间。
 - 然后再删掉整个表。
- ExecuteEngine :: ExecuteShowIndexes: 执行形如 show indexes 的 SQL 语句

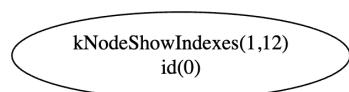


Figure 18: ExecuteShowIndexes 要执行的语法树

src/executor/execute_engine.cpp

```
1 dberr_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast,
2 ExecuteContext *context) {
3 #ifdef ENABLE_EXECUTE_DEBUG
4     LOG(INFO) << "ExecuteShowIndexes" << std::endl;
5 #endif
6     if (current_db_.empty()) {
7         cout << "No database selected" << endl;
8         return DB_FAILED;
9     }
10    vector<string> table_names;
11    vector<TableInfo *> tables;
12    vector<IndexInfo *> indexes;
13    map<string, vector<IndexInfo *>> table_name2indexes;
14    string index_in_db("All Indexes");
15    uint max_width = index_in_db.length();
16
17    if (dbs_[current_db_]->catalog_mgr_->GetTables(tables) ==
18 DB_FAILED) {
19        cout << "Empty set (0.00 sec)" << endl;
20        return DB_FAILED;
21    }
22    for (const auto &table : tables)
23        table_names.push_back(table->GetTableName());
24    bool has_index = false;
25    for (const auto &table_name : table_names) {
26        if (dbs_[current_db_]->catalog_mgr_-
27 >GetTableIndexes(table_name, indexes) == DB_SUCCESS) {
28            if (indexes.size() > 0)
29                has_index = true;
30            table_name2indexes[table_name] = indexes;
31            if (string("Index in " + table_name).length() >
32 max_width)
33                max_width = string("Index in " + table_name).length();
34            for (const auto &index : indexes) {
35                if (index->GetIndexName().length() > max_width)
36                    max_width = index->GetIndexName().length();
37            }
38        // no index found
39        if (!has_index) {
40            cout << "Empty set (0.00 sec)" << endl;
41            return DB_FAILED;
42        }
43    }
44}
```

```

43
44     // formatting
45     cout << "+" << setfill('-') << setw(max_width + 2) << ""
46         << "+" << endl;
47     cout << "|" << std::left << setfill(' ') << setw(max_width)
48         << index_in_db << " |" << endl;
49     cout << "+" << setfill('-') << setw(max_width + 2) << ""
50         << "+" << endl;
51     for (const auto &t2i : table_name2indexes) {
52         cout << "| " << std::left << setfill(' ') <<
53             setw(max_width) << "Index in " + t2i.first << " |" << endl;
54         cout << "+" << setfill('-') << setw(max_width + 2) << ""
55             << "+" << endl;
56         for (const auto &index : t2i.second) {
57             cout << "| " << std::left << setfill(' ') <<
58                 setw(max_width) << index->GetIndexName() << " |" << endl;
59         }
60     cout << "+" << setfill('-') << setw(max_width + 2) << ""
61         << "+" << endl;
62     }
63
64     return DB_SUCCESS;
65 }
```

- 该方法会打印数据库每个表的全部索引（因为没有指定显示哪个表的索引）。
- 打印格式参照已经实现了的 ExecuteShowTables 方法。
- ExecuteEngine::ExecuteCreateIndex: 执行形如 create index ... on ... using ... 的 SQL 语句

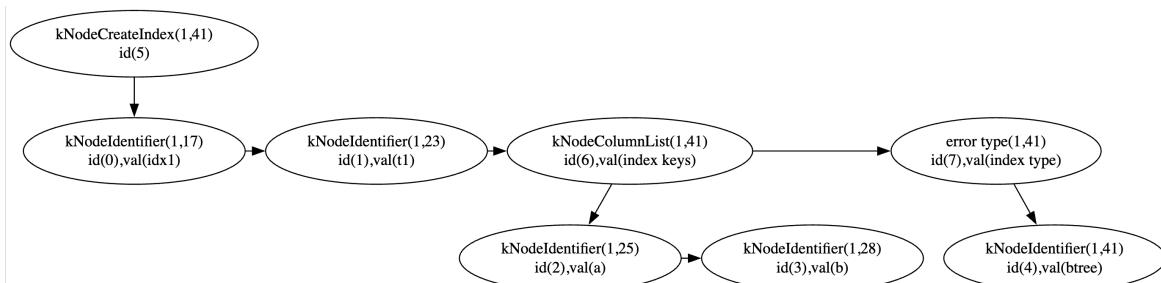


Figure 19: ExecuteCreateIndex 要执行的语法树

src/executor/execute_engine.cpp

```

1 dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast,
2                                         ExecuteContext *context) {
```

```

2 #ifdef ENABLE_EXECUTE_DEBUG
3     LOG(INFO) << "ExecuteCreateIndex" << std::endl;
4 #endif
5     if (current_db_.empty()) {
6         cout << "No database selected" << endl;
7         return DB_FAILED;
8     }
9
10    auto catalog = context->GetCatalog();
11    string index_name = ast->child_->val_;
12    string table_name = ast->child_->next_->val_;
13    vector<string> index_keys;
14    auto index_key_node = ast->child_->next_->next_->child_;
15
16    while (index_key_node) {
17        string column_name = index_key_node->val_;
18        index_keys.push_back(column_name);
19        index_key_node = index_key_node->next_;
20    }
21
22    string index_type = "";
23    if (ast->child_->next_->next_->next_)
24        index_type = ast->child_->next_->next_->next_->val_;
25
26    IndexInfo *index_info;
27    if (catalog->CreateIndex(table_name, index_name,
28 index_keys, nullptr, index_info, index_type) != DB_SUCCESS)
29    {
30        return DB_FAILED;
31    }
32    return DB_SUCCESS;
33 }
```

- 没有什么特别之处，就是根据语法树的节点内容提取建立索引用到的列名、索引类型等信息，最后使用 CatalogManager::CreateIndex 方法创建索引即可。
- ExecuteEngine::ExecuteDropIndex: 执行形如 drop index ... 的 SQL 语句

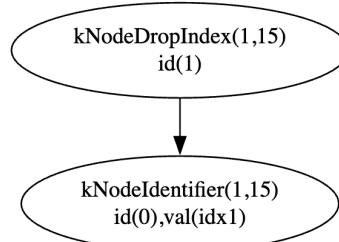


Figure 20: ExecuteDropIndex 要执行的语法树

src/executor/execute_engine.cpp

```

1  dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast,
2  ExecuteContext *context) {
3  #ifdef ENABLE_EXECUTE_DEBUG
4      LOG(INFO) << "ExecuteDropIndex" << std::endl;
5  #endif
6      if (current_db_.empty()) {
7          cout << "No database selected" << endl;
8          return DB_FAILED;
9      }
10     auto catalog = context->GetCatalog();
11     string index_name = ast->child_->val_;
12     string table_name = "";
13     vector<TableInfo *> tables;
14     vector<IndexInfo *> indexes;
15     vector<string> table_names;
16
17     if (dbs_[current_db_]->catalog_mgr_->GetTables(tables) ==
18         DB_FAILED)
19         return DB_FAILED;
20     for (const auto &table : tables)
21         table_names.push_back(table->GetTableName());
22
23     for (const auto &tn : table_names) {
24         // find the corresponding table name of the index
25         if (dbs_[current_db_]->catalog_mgr_->GetTableIndexes(tn,
26             indexes) == DB_SUCCESS) {
27             for (const auto &index : indexes) {
28                 if (index->GetIndexName() == index_name) {
29                     table_name = tn;
30                     break;
31                 }
32             }
33         }
34     }
35 }
```

```

34     if (catalog->DropIndex(table_name, index_name) != DB_SUCCESS) {
35         cout << "Failed to drop the index!" << endl;
36         return DB_FAILED;
37     }
38     return DB_SUCCESS;
39 }

```

- 由于这个语句没有指定表明，所以首先要做的就是找到这个索引出现在哪个表中。
- 个人认为这种形式的删除索引语句设计不太合理。应该像 MySQL 等主流 DBMS 那样采用形如 `drop index idx on table tb;` 这样的语句。由于时间关系，小组并没重新设计该语句的实现。
- 找到表名后就可以调用 `CatalogManager::DropIndex` 方法来删索引了。
- `ExecuteEngine::ExecuteExecfile`: 执行形如 `execfile ...` 的 SQL 语句

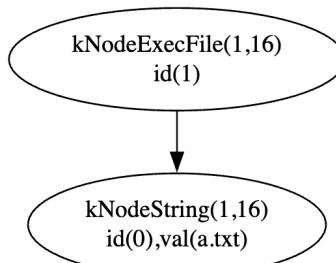


Figure 21: `ExecuteExecfile` 要执行的语法树

src/executor/execute_engine.cpp

```

1  dberr_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast,
2  ExecuteContext *context) {
3 #ifdef ENABLE_EXECUTE_DEBUG
4     LOG(INFO) << "ExecuteExecfile" << std::endl;
5 #endif
6     if (current_db_.empty()) {
7         cout << "No database selected" << endl;
8         return DB_FAILED;
9     }
10    string file_name = ast->child_->val_;
11    fstream file;
12    file.open(file_name);
13    if (!file.is_open()) {
14        cout << "File not found" << endl;

```

```

15     return DB_FAILED;
16 }
17
18 // for print syntax tree
19 TreeFileManagers syntax_tree_file_mngr("syntax_tree_");
20 uint32_t syntax_tree_id = 0;
21 char ch;
22 const int buf_size = 1024;
23 char sql[buf_size];
24 int i;
25
26 // start timing
27 auto start_time = chrono::system_clock::now();
28 while (!file.eof()) {
29     i = 0;
30     while (!file.eof() && (ch = file.get()) != ';')
31         sql[i++] = ch;
32
33     if (file.eof())
34         continue;
35
36     sql[i++] = ch;      // ;
37     sql[i] = '\0';      // don't forget it!
38
39     cout << sql << endl;
40     // create buffer for sql input
41     YY_BUFFER_STATE bp = yy_scan_string(sql);
42
43     if (bp == nullptr) {
44         LOG(ERROR) << "Failed to create yy buffer state."
45         << std::endl;
46         exit(1);
47     }
48     yy_switch_to_buffer(bp);
49
50     // init parser module
51     MinisqlParserInit();
52
53     // parse
54     yyparse();
55
56     // parse result handle
57     if (MinisqlParserGetError()) {
58         // error
59         printf("%s\n", MinisqlParserErrorMessage());
60     }
61     auto result = Execute(MinisqlGetParserRootNode());
62

```

```

63     // clean memory after parse
64     MinisqlParserFinish();
65     yy_delete_buffer(bp);
66     yylex_destroy();
67
68     // quit condition
69     ExecuteInformation(result);
70     if (result == DB_QUIT) {
71         break;
72     }
73 }
74 // end timing
75 auto stop_time = chrono::system_clock::now();
76 double duration_time =
77     double((chrono::duration_cast<chrono::milliseconds>(stop_time
78 - start_time)).count());
79 cout << "Execfile finished in " << duration_time << " ms"
80 << endl;
81 file.close();
82
83     return DB_SUCCESS;
84 }
```

- 这里的代码实现基本仿照 `main.cpp` 来编写的。两者最大的区别在于 `main.cpp` 接收的是标准输入 `stdin`, 而本方法接收的是文件输入流 `ifstream`。
- 根据实验要求, 该方法还需要具备计时功能。计时的始末位于读取文件内 SQL 语句的循环的前后 (仿照着其他的 executor 编写的)。
- `ExecuteEngine::ExecuteQuit`: 执行 `quit` 语句

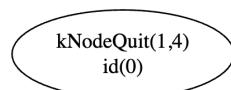


Figure 22: `ExecuteQuit` 要执行的语法树

src/executor/execute_engine.cpp

```

1 dberr_t ExecuteEngine::ExecuteQuit(pSyntaxNode ast,
2 ExecuteContext *context) {
3 #ifdef ENABLE_EXECUTE_DEBUG
4     LOG(INFO) << "ExecuteQuit" << std::endl;
5 #endif
6     current_db_.clear();
7     return DB_QUIT;
8 }
```

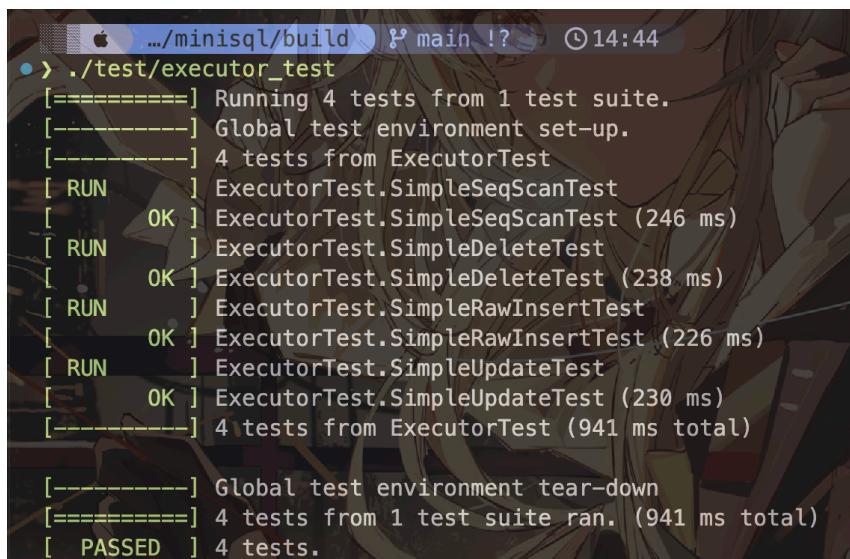
7 }

2.5.3 测试

先来看已有的测试代码：

- `SimpleSeqScanTest`: 检验程序是否能正确执行简单的顺序扫描，也就是 `select` 语句默认采用的方式。
- `SimpleDeleteTest`: 检验程序能否正确删除表中的指定记录；在此之前还创建了索引。
- `SimpleRawInsertTest`: 检验程序能否成功地向表插入新记录。
- `SimpleUpdateTest`: 检验程序能否成功更新表中已有的记录。

测试结果如下：



```
.../minysql/build $ main !? ① 14:44
$ ./test/executor_test
[=====] Running 4 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 4 tests from ExecutorTest
[RUN    ] ExecutorTest.SimpleSeqScanTest
[OK     ] ExecutorTest.SimpleSeqScanTest (246 ms)
[RUN    ] ExecutorTest.SimpleDeleteTest
[OK     ] ExecutorTest.SimpleDeleteTest (238 ms)
[RUN    ] ExecutorTest.SimpleRawInsertTest
[OK     ] ExecutorTest.SimpleRawInsertTest (226 ms)
[RUN    ] ExecutorTest.SimpleUpdateTest
[OK     ] ExecutorTest.SimpleUpdateTest (230 ms)
[-----] 4 tests from ExecutorTest (941 ms total)

[-----] Global test environment tear-down
[=====] 4 tests from 1 test suite ran. (941 ms total)
[PASSED ] 4 tests.
```

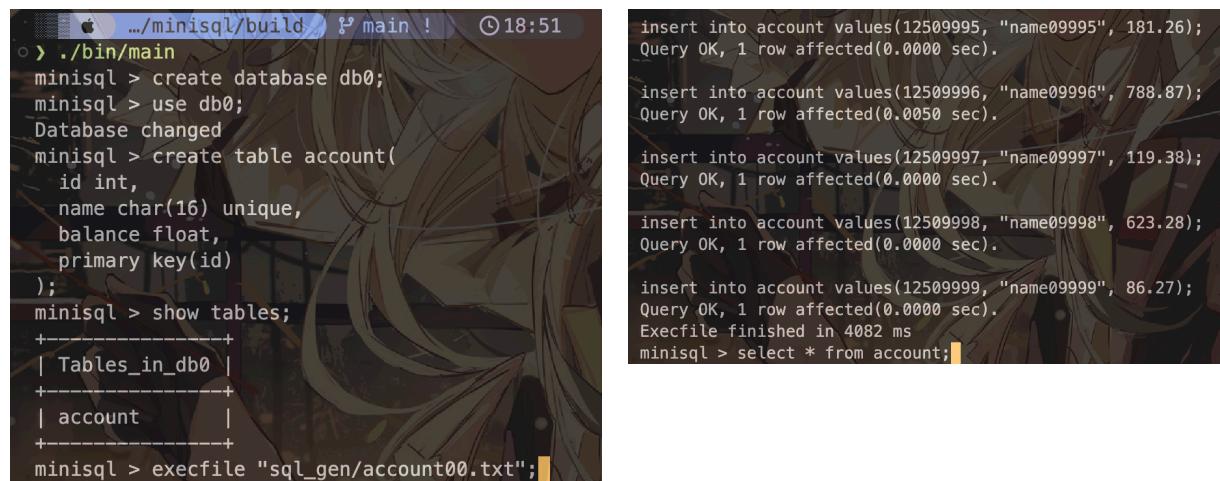
Figure 23: 通过 `executor_test` 的测试

我们认为已有的测试代码主要对 `Executor` 模块中除了 `execute_engine` 外的其他子模块进行测试，并没有覆盖到我们自己实现的部分。实际上，我们可以通过直接执行主程序来验证 `Executor` 是否能够正确执行所有可能的 SQL 语句。于是，我们测试了以下 SQL 语句，观察执行结果是否符合预期：

```
1 create database db0;
2 use db0;
3 create table account(
4     id int,
5     name char(16) unique,
6     balance float,
```

```
7     primary key(id)
8 );
9 show tables;
10 execfile "sql_gen/account00.txt";
11 select * from account;
12 create index idx01 on account(name);
13 show indexes;
14 drop index idx01;
15 show indexes;
16 delete from account;
17 select * from account;
18 drop table account;
19 show tables;
20 quit;
```

测试结果如下：



```
.../minisql/build ➜ main ! ⑧ 18:51
❯ ./bin/main
minisql > create database db0;
minisql > use db0;
Database changed
minisql > create table account(
    id int,
    name char(16) unique,
    balance float,
    primary key(id)
);
minisql > show tables;
+-----+
| Tables_in_db0 |
+-----+
| account      |
+-----+
minisql > execfile "sql_gen/account00.txt";
insert into account values(12509995, "name09995", 181.26);
Query OK, 1 row affected(0.0000 sec).
insert into account values(12509996, "name09996", 788.87);
Query OK, 1 row affected(0.0050 sec).
insert into account values(12509997, "name09997", 119.38);
Query OK, 1 row affected(0.0000 sec).
insert into account values(12509998, "name09998", 623.28);
Query OK, 1 row affected(0.0000 sec).
insert into account values(12509999, "name09999", 86.27);
Query OK, 1 row affected(0.0000 sec).
Execfile finished in 4082 ms
minisql > select * from account;
```



```
| 12509999 | name09999 | 86.269997 |
+-----+
10000 row in set(0.0630 sec).
minysql > create index idx01 on account(name);
minysql > show indexes;
+-----+
| All Indexes |
+-----+
| Index in account |
+-----+
| pk_account |
| uk_account_name |
| idx01 |
+-----+
minysql > drop index idx01;
minysql > show indexes;
+-----+
| All Indexes |
+-----+
| Index in account |
+-----+
| pk_account |
| uk_account_name |
+-----+
minysql > delete from account;
Query OK, 10000 row affected(2.8020 sec).
minysql > select * from account;
W20250526 18:53:51.944895 2342023 table_heap.cpp:239] Failed to find a valid tuple when get Begin iterator
W20250526 18:53:51.946959 2342023 table_heap.cpp:239] Failed to find a valid tuple when get Begin iterator
Empty set(0.0020 sec).
minysql > drop table account;
minysql > show tables;
+-----+
| Tables_in_db0 |
+-----+
minysql > quit;
Bye.
```

Figure 24: 通过主程序来测试 Executor 模块

可以看到，执行结果均符合预期，这证明了 Executor 代码实现的正确性。

2.6 Recovery Manager

2.6.1 原理和代码实现

模块 6 实际上完全独立于整个项目，可以认为是一个单独的小实验。

在这个实验中，数据库、日志的内容都存储在内存中而没有落盘。数据库是使用哈希表模拟的，日志也没有统一管理，每条日志都是独立存在，只是通过日志号从逻辑上串联起来。

在这个实验中，我们首先需要实现创建 6 种不同类型的日志，并将同一事务的通过 `prev_lsn_` 连接起来。另外就是基于日志，构建基于 `CheckPoint` 类实现的 `RecoveryManager` 类。但实际上在 `RecoveryManager` 类，只需要完成独立的 `RedoPhase()` 与 `UndoPhase()`。下面将分别介绍这两个任务。

2.6.1.1 创建日志

首先需要完善 LogRec 结构体的定义，我在这里增加了所属事务的事务号，以及修改前后的键值对信息。

```
src/include/recovery/log_rec.h
```

```
1 struct LogRec {
2     LogRec() = default;
3
4     LogRecType type_{LogRecType::kInvalid};
5     lsn_t lsn_{INVALID_LSN};
6     lsn_t prev_lsn_{INVALID_LSN};
7     txn_id_t txn_id_;
8     std::pair<KeyType, ValType> old_rec_;
9     std::pair<KeyType, ValType> new_rec_;
10
11    /* used for testing only */
12    static std::unordered_map<txn_id_t, lsn_t> prev_lsn_map_;
13    static lsn_t next_lsn_;
14};
```

不同类型的日志虽然具体构造上有些许不同，但整体上相似。以 CreateUpdateLog() 为例，首先需要从 prev_lsn_map_ 中获取所属事务中的上一条日志，并更新 prev_lsn_map_，随后依据 next_lsn_ 设置自身的 lsn_，最后根据传入的信息更新 old_rec_ 与 new_rec_。

```
src/include/recovery/log_rec.h
```

```
1 static LogRecPtr CreateUpdateLog(txn_id_t txn_id, KeyType
2     old_key, ValType old_val, KeyType new_key, ValType new_val) {
3     auto prev = LogRec::prev_lsn_map_.find(txn_id);
4     if (prev == LogRec::prev_lsn_map_.end()) {
5         throw "Invalid Txn id";
6         return nullptr;
7     }
8     LogRecPtr ptr = std::make_shared<LogRec>();
9     ptr->type_ = LogRecType::kUpdate;
10    ptr->prev_lsn_ = prev->second;
11    ptr->lsn_ = LogRec::next_lsn_++;
12    ptr->txn_id_ = txn_id;
13    LogRec::prev_lsn_map_[txn_id] = ptr->lsn_;
14
15    ptr->old_rec_ = make_pair(old_key, old_val);
16    ptr->new_rec_ = make_pair(new_key, new_val);
```

```
17     return ptr;
18 }
```

其他类型的日志区别列举如下：

- CreateInsertLog(): 只需要更新 new_rec_
- CreateDeleteLog(): 只需要更新 old_rec_
- CreateBeginLog(): 错误条件改为查询 prev_lsn_map_ 出现非空结果，同时 ptr→prev_lsn_ = INVALID_LSN
- CreateCommitLog() 与 CreateAbortLog(): 从 prev_lsn_map_ 中除去当前事务号

2.6.1.2 恢复系统

恢复系统的原理在《数据库系统》课程中已经有详细介绍，这里只是实现一个较为简易的版本。在框架中，一共需要实现三个方法：

- 第一个方法 Init() 是依据传入的检查点的信息初始化 RecoveryManager，只需要给对应成员赋值即可。
- 第二个方法 RedoPhase()，从检查点所处的 lsn 向后依次 redo 每一条 log，依照 log 类型相应的更新 data_ 与 active_txns_，具体实现较为简单，这里也就不作呈现。
- 第三个方法 UndoPhase()，从 active_txns_ 中获取活跃的事务的 txn_id，然后依次 Undo 即可。但由于在 RedoPhase() 中，遇到 Abort 也需要 Undo 对应的事务，所以我增加了 UndoTxn() 这一辅助方法，根据指令提供的 prev_lsn_ 依次做相反操作即可实现，代码如下：

src/include/recovery/recovery_manager.h

```
1 void UndoTxn(txn_id_t txn_id) {
2     auto active_txn = active_txns_.find(txn_id);
3     if(active_txn == active_txns_.end()) return;
4     auto it = log_recs_.find(active_txn->second);
5
6
7     while (it != log_recs_.end()) {
8         const auto &rec = it->second;
9         switch (rec->type_) {
10             case LogRecType::kInsert:
11                 data_.erase(rec->new_rec_.first);
12                 break;
```

```

13     case LogRecType::kUpdate:
14         if (rec->new_rec_.first == rec->old_rec_.first) {
15             data_[rec->new_rec_.first] = rec->old_rec_.second;
16         } else {
17             data_[rec->old_rec_.first] = rec->old_rec_.second;
18             data_.erase(rec->new_rec_.first);
19         }
20         break;
21     case LogRecType::kDelete:
22         data_[rec->old_rec_.first] = rec->old_rec_.second;
23         break;
24     default:
25         break;
26     }
27     it = log_recs_.find(rec->prev_lsn_);
28 }
29 }
```

2.6.2 测试

虽然实验报告上要求完成 `recovery_manager_test.cpp`, 但实际上测试代码已经十分全面的测试了实现的所有功能, 所以我没有增加新的测试。

测试结果如图所示:

```

/mnt/d/nCoding/cpp/2025-04/minysql/build | on au :4 *1
> ./test/recovery_manager_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from RecoveryManagerTest
[RUN    ] RecoveryManagerTest.RecoveryTest
[OK     ] RecoveryManagerTest.RecoveryTest (0 ms)
[-----] 1 test from RecoveryManagerTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[PASSED ] 1 test.
```

Figure 25: #6 测试结果

2.6.3 思考题

题目

本模块中, 为了简化实验难度, 我们将 Recovery Manager 模块独立出来。如果不独立出来, 真正做到数据库在任何时候断电都能恢复, 同时支持事务的回滚, Recovery Manager 应该怎样设计呢? 此外, CheckPoint 机制应该怎样设计呢?

可以采用 ARIES 算法重新设计 Recovery Manager, 所以这里先复述一下 ARIES 算法的原理。

2.6.3.1 ARIES 算法

ARIES 算法的主要思想是：

- WAL with STEAL/NO-FORCE
- Repeating History During Redo: 重做阶段重现崩溃前的状态
- Logging changes during undo: 在事务回滚做 Undo 的时候, 要为 Undo 产生的更新操作记录

基于这些思想, Recovery Manager 在恢复数据库时, 设计了一些核心组件:

- ATT (活跃事务表) : 存储活跃事务 (已开始但未提交/回滚) 的信息, 记录每一个活跃事务的状态与它们的 LastLSN, 但每条日志都需要记录该事务的 preLSN
- DPT (脏页表) : 存储脏页信息, 记录使数据页变脏的第一条与最后一条日志的 LSN, 同时, 但每个数据页本身也记录 pageLSN (即使最后一条修改数据页的日志的 LSN)
- checkpoint (检查点) : 记录检查点时刻的日志的 LSN, 并保存当时的 ATT 与 DPT。

有了这些组件, ARIES 将恢复分为三个阶段:

- **分析阶段:** 确定崩溃前的系统状态
 - 从硬盘中的日志找到最后一个已完成刷盘的 checkpoint, 并获取当时的 ATT (活跃事务表) 与 DPT (脏页表);
 - 从 checkpoint 的 LSN 开始, 遍历之后所有的日志, 恢复崩溃时的 ATT 与 DPT, 如果遇到 TXN-END 类型日志, 则将事务移出 LSN;
 - 分析完成后, 设置重做阶段的初始 LSN (即 DPT 中找到最小的 recLSN)
- **重做阶段:** 完全再现崩溃前的历史
 - 从 DPT 中找到最小的 recLSN, 开始依次执行每一条日志, 但对于需要更新数据页的日志, 只有 LSN 大于 pageLSN 才能执行;
 - 为每一个 COMMITTED 状态的事务写下 TXN-END, 移出 ATT;
- **撤销阶段:** 回滚所有未提交事务
 - 找到 ATT 中最大的 LastLSN, 撤销它对应的事务, 并记录 CLR, 撤销完成后把该事务从 ATT 中移除;
 - 重复执行上一步, 直到 ATT 为空。

通过上面的设计, 我们可以保证数据库在任何时刻断电都可以恢复, 同时支持事务的回滚。

2.6.3.2 程序设计

如果想要实现上面的算法，主要要完成 Log Manager 与 Recovery Manager 部分的设计：

- **CheckPoint Manager:** 由于检查点与日志区别较大，所以我认为需要设计 CheckPoint Manager，进而实现对 CheckPoint 的管理。在这一组件中需要实现：
 - 检查点类的定义，与创建检查点方法的实现；
 - 提供创建 ATT 与 DPT 快照的方法，如果有锁的存在，为了减少长时间锁住 ATT 与 DPT，可以将 ATT 与 DPT 分成多个部分，分批创建快照；
- **Log Manager:** Log Manager 是为了实现 WAL 策略的主要组件，为了实现这一组件首先需要完善的底层基础设施：
 - Disk Manager：类似对数据页的处理，实现对日志页的分配、存储、读入与写入操作，但需要为日志页提供强制刷盘的方法；
 - 数据页：需要给每个数据页增加 PageLSN 成员，并提供读取、修改的方法；
 - 日志：我们应该按照 ARIES 算法的要求，定义日志类的成员与方法，提供创建对应类型日志的方法，以及序列化及反序列化日志数据的方法；
 - 事务：虽然我们的框架中没有，但应该在事务执行操作、开始、结束时都触发对应日志的生成；

有了这些基础设施还有之前的 CheckPoint Manager，我们需要在 Log Manager 中实现如下的功能：

- 维护基础架构：ATT、DPT、flushedLSN 与 nextLSN；
- 提供生成日志并修改日志页的接口；
- 维护日志页缓存池，并实现日志页的替换策略（因为日志是顺序写入，所以不需要使用 LRU 策略，可以采用顺序替换），同时含需要提供刷盘方法；
- 完成生成检查点的机制，比如定时生成、在刷盘一定数量日志后生成，并负责检查点的调用。

基于以上的功能，我们还需要修改事务管理器和 Buffer Pool Manager 相关逻辑：

- 事务管理器需要事务在执行特定操作时调用 Log Manager 中的方法，生成对应的日志；
- Buffer Pool Manager 需要在数据页刷盘前确保日志已经刷盘，如果没有刷盘，则需要调用 Log Manager 刷盘日志，同时也需要增加更新 PageLSN 的相关逻辑。
- **Recovery Manager:** 主要需要实现在崩溃后恢复系统需要的方法，具体如下：
 - 读取日志，找到最后的检查点并读取 ATT 与 DPT 快照的方法；

- 依照 ARIES 算法实现 Analyse(), Redo() 与 Undo() 三大基本操作，并封装在 Recovery() 中。

2.7 Lock Manager

2.7.1 原理

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共享(shared)锁和独占(exclusive)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 里定义了两个类：

- `LockRequest`: 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`。
- `LockRequestQueue`: 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

我们需要完善 `LockManager` 类中的这些函数：

- `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` 请求升级 id 为 `RID` 的数据记录上的共享锁至独占锁，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)`: 释放事务 `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 `kGrowing` 阶段变为 `kShrinking` 阶段。
 - `LockPrepare(Txn, RID)`: 检测 `Txn` 的状态是否符合预期，并在 `lock_table_` 里创建 `RID` 和对应的队列。
 - `CheckAbort(Txn, LockRequestQueue)`: 检查 `Txn` 的状态是否是 `kAborted`，如果是，做出相应的操作。
-

此外，锁管理器还应具备在后台运行死锁检测的能力，以中止阻塞事务。具体来说，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。

2.7.2 代码实现

由于前面已经详细介绍过每个函数的作用了，这里就不再赘述，直接列出代码：

```
src/concurrency/lock_manager.cpp
```

```

1  bool LockManager::LockShared(Txn *txn, const RowId &rid) {
2      // 如果隔离级别是 READ_UNCOMMITTED，不允许加共享锁
3      if (txn->GetIsolationLevel() == 
4          IsolationLevel::kReadUncommitted) {
5          txn->SetState(TxnState::kAborted);
6          throw TxnAbortException(txn->GetTxnId(),
7          AbortReason::kLockSharedOnReadUncommitted);

```

```

6     }
7
8     std::unique_lock<std::mutex> lk(latch_);
9     LockPrepare(txn, rid);
10    auto &req_queue = lock_table_.at(rid);
11
12    // 检查事务是否已经持有该记录的锁
13    auto iter = req_queue.req_list_iter_map_.find(txn-
14        >GetTxnId());
15    if (iter != req_queue.req_list_iter_map_.end()) {
16        if (iter->second->granted_ != LockMode::kNone) {
17            return true; // 已经持有锁，直接返回
18        }
19    } else {
20        // 添加新的锁请求
21        req_queue.EmplaceLockRequest(txn->GetTxnId(),
22            LockMode::kShared);
23        iter = req_queue.req_list_iter_map_.find(txn-
24        >GetTxnId());
25    }
26
27    // 等待直到可以获取共享锁
28    while (true) {
29        CheckAbort(txn, req_queue);
30
31        // 如果没有写锁且没有正在进行的锁升级，可以获取共享锁
32        if (!req_queue.is_writing_ && !req_queue.is_upgrading_)
33        {
34            iter->second->granted_ = LockMode::kShared;
35            req_queue.sharing_cnt_++;
36            txn->GetSharedLockSet().insert(rid);
37            return true;
38        }
39
40        // 等待其他事务释放锁
41        req_queue.cv_.wait(lk);
42    }
43
44    return false;
45
46    bool LockManager::LockExclusive(Txn *txn, const RowId &rid) {
47        std::unique_lock<std::mutex> lk(latch_);
48        LockPrepare(txn, rid);
49        auto &req_queue = lock_table_.at(rid);
50
51        // 检查事务是否已经持有该记录的锁
52
53        if (req_queue.is_writing_ || req_queue.is_upgrading_) {
54            return false;
55        }
56
57        if (req_queue.req_list_iter_map_.find(txn-
58            >GetTxnId()) != req_queue.req_list_iter_map_.end()) {
59            return false;
60        }
61
62        req_queue.EmplaceLockRequest(txn->GetTxnId(),
63            LockMode::kExclusive);
64        req_queue.is_writing_ = true;
65
66        return true;
67    }
68
69
```

```

        auto iter = req_queue.req_list_iter_map_.find(txn-
49 >GetTxnId());
50     if (iter != req_queue.req_list_iter_map_.end()) {
51         if (iter->second->granted_ == LockMode::kExclusive) {
52             return true; // 已经持有独占锁，直接返回
53         }
54     } else {
55         // 添加新的锁请求
56         req_queue.EmplaceLockRequest(txn->GetTxnId(),
57                                         LockMode::kExclusive);
58         iter = req_queue.req_list_iter_map_.find(txn-
59 >GetTxnId());
60     }
61
62     // 等待直到可以获取独占锁
63     while (true) {
64         CheckAbort(txn, req_queue);
65
66         // 如果没有其他锁，可以获取独占锁
67         if (!req_queue.is_writing_ && req_queue.sharing_cnt_-
68 = 0) {
69             iter->second->granted_ = LockMode::kExclusive;
70             req_queue.is_writing_ = true;
71             txn->GetExclusiveLockSet().insert(rid);
72             return true;
73         }
74
75         // 等待其他事务释放锁
76         req_queue.cv_.wait(lk);
77     }
78
79     bool LockManager::LockUpgrade(Txn *txn, const RowId &rid) {
80         std::unique_lock<std::mutex> lk(latch_);
81         auto &req_queue = lock_table_.at(rid);
82
83         // 检查是否已经有其他事务在进行锁升级
84         if (req_queue.is_upgrading_) {
85             txn->SetState(TxnState::kAborted);
86             throw TxnAbortException(txn->GetTxnId(),
87                                     AbortReason::kUpgradeConflict);
88         }
89
90         // 检查事务是否持有共享锁

```

```

        auto iter = req_queue.req_list_iter_map_.find(txn-
90 >GetTxnId());
        if (iter == req_queue.req_list_iter_map_.end() || iter-
91 >second->granted_ != LockMode::kShared) {
92     txn->SetState(TxnState::kAborted);
93     throw TxnAbortException(txn->GetTxnId(),
94 AbortReason::kLockOnShrinking);
95 }
96
97 // 标记正在进行锁升级
98 req_queue.is_upgrading_ = true;
99 iter->second->lock_mode_ = LockMode::kExclusive;
100
101 // 等待直到可以获取独占锁
102 while (true) {
103     CheckAbort(txn, req_queue);
104
105     // 如果只剩下当前事务持有共享锁，可以升级为独占锁
106     if (!req_queue.is_writing_ && req_queue.sharing_cnt_-
107 = 1) {
108         iter->second->granted_ = LockMode::kExclusive;
109         req_queue.is_writing_ = true;
110         req_queue.is_upgrading_ = false;
111         req_queue.sharing_cnt_--;
112
113         // 更新事务的锁集合
114         txn->GetSharedLockSet().erase(rid);
115         txn->GetExclusiveLockSet().insert(rid);
116
117         return true;
118     }
119
120     // 等待其他事务释放锁
121     req_queue.cv_.wait(lk);
122
123     return false;
124 }
125
126 bool LockManager::Unlock(Txn *txn, const RowId &rid) {
127     std::unique_lock<std::mutex> lk(latch_);
128
129     // 如果事务状态是 GROWING，则转换为 SHRINKING
130     if (txn->GetState() == TxnState::kGrowing) {
131         txn->SetState(TxnState::kShrinking);
132     } else if (txn->GetState() == TxnState::kShrinking) {
133         txn->SetState(TxnState::kAborted);

```

```

133             throw TxnAbortException(txn->GetTxnId(),
134             AbortReason::kUnlockOnShrinking);
135         }
136
137         auto &req_queue = lock_table_.at(rid);
138         auto iter = req_queue.req_list_iter_map_.find(txn-
139             >GetTxnId());
140         if (iter == req_queue.req_list_iter_map_.end()) {
141             return false; // 事务没有持有该记录的锁
142         }
143
144         // 根据锁类型更新计数器和标志
145         if (iter->second->granted_ == LockMode::kShared) {
146             req_queue.sharing_cnt_--;
147             txn->GetSharedLockSet().erase(rid);
148         } else if (iter->second->granted_ == LockMode::kExclusive)
149         {
150             req_queue.is_writing_ = false;
151             txn->GetExclusiveLockSet().erase(rid);
152         }
153
154         // 从请求队列中移除该事务的请求
155         req_queue.EraseLockRequest(txn->GetTxnId());
156
157         // 通知其他等待的事务
158         req_queue.cv_.notify_all();
159
160     return true;
161 }
162
163 void LockManager::LockPrepare(Txn *txn, const RowId &rid) {
164     // 检查事务状态，如果是 SHRINKING，则抛出异常
165     if (txn->GetState() == TxnState::kShrinking) {
166         txn->SetState(TxnState::kAborted);
167         throw TxnAbortException(txn->GetTxnId(),
168             AbortReason::kLockOnShrinking);
169     }
170
171     // 在 lock_table_ 中创建 rid 对应的队列（如果不存在）
172     if (lock_table_.find(rid) == lock_table_.end()) {
173         lock_table_.try_emplace(rid);
174     }
175
176     void LockManager::CheckAbort(Txn *txn,
177         LockManager::LockRequestQueue &req_queue) {
178         // 检查事务状态是否为 ABORTED

```

```

175     if (txn->GetState() == TxnState::kAborted) {
176         // 从请求队列中删除该事务的请求
177         req_queue.EraseLockRequest(txn->GetTxnId());
178         // 通知其他等待的事务
179         req_queue.cv_.notify_all();
180         // 抛出异常
181         throw TxnAbortException(txn->GetTxnId(),
182             AbortReason::kDeadlock);
183     }
184 }
185 void LockManager::AddEdge(txn_id_t t1, txn_id_t t2) {
186     waits_for_[t1].insert(t2);
187     visited_set_.clear();
188     while (!visited_path_.empty())
189         visited_path_.pop();
190 }
191
192 void LockManager::RemoveEdge(txn_id_t t1, txn_id_t t2) {
193     waits_for_[t1].erase(t2);
194     visited_set_.clear();
195     while (!visited_path_.empty())
196         visited_path_.pop();
197 }
198
199 bool LockManager::HasCycle(txn_id_t &newest_tid_in_cycle) {
200     auto& tid = newest_tid_in_cycle; // avoid
201     using lengthy name
202
203     // find the cycle!
204     if (visited_set_.count(tid) > 0) {
205         auto youngest_tid = tid;
206         for (const auto& elem : visited_set_) {
207             if (elem < youngest_tid) {
208                 youngest_tid = elem;
209             }
210         }
211         tid = youngest_tid; // set the youngest tid
212         return true;
213     }
214
215     // tid is visited
216     visited_set_.insert(tid);
217     visited_path_.push(tid);
218
219     // dfs
220     for (const auto& next_tid : waits_for_[tid]) {

```

```

220         txn_id_t temp_tid = next_tid; // Create a non-
221     const copy
222         if (HasCycle(temp_tid)) {
223             newest_tid_in_cycle = temp_tid;
224             return true;
225         }
226
227         // backtrack
228         visited_path_.pop();
229         visited_set_.erase(tid);
230
231         return false;
232     }
233
234     void LockManager::RunCycleDetection() {
235         while (enable_cycle_detection_) {
236             waits_for_.clear();
237
238             // create wait-for graph from lock table
239             for (const auto &kv : lock_table_) {
240                 auto &req_list = kv.second.req_list_;
241                 // tranverse the req_list in reverse order
242                 for (auto ri = req_list.rbegin(); ri != req_list.rend(); ++ri) {
243                     if (ri->granted_ == LockMode::kNone)
244                         break;
245                     for (auto rj = ri; rj != req_list.rend(); rj+
246 +) {
247                         if ((rj->granted_ == LockMode::kNone) &&
248                             ((ri->lock_mode_ == LockMode::kShared
249                             && rj->lock_mode_ == LockMode::kExclusive) || (ri->lock_mode_
250                             == LockMode::kExclusive))) {
251                             AddEdge(rj->txn_id_, ri->txn_id_);
252                         }
253                     }
254
255                     // clear records of visited nodes
256                     visited_set_.clear();
257                     while (!visited_path_.empty()) visited_path_.pop();
258
259                     // deadlock detection
260                     txn_id_t newest_tid_in_cycle = INVALID_TXN_ID;
261                     if (HasCycle(newest_tid_in_cycle)) {
262                         txn_mgr_->Abort(txn_mgr_-
263 >GetTransaction(newest_tid_in_cycle));

```

```

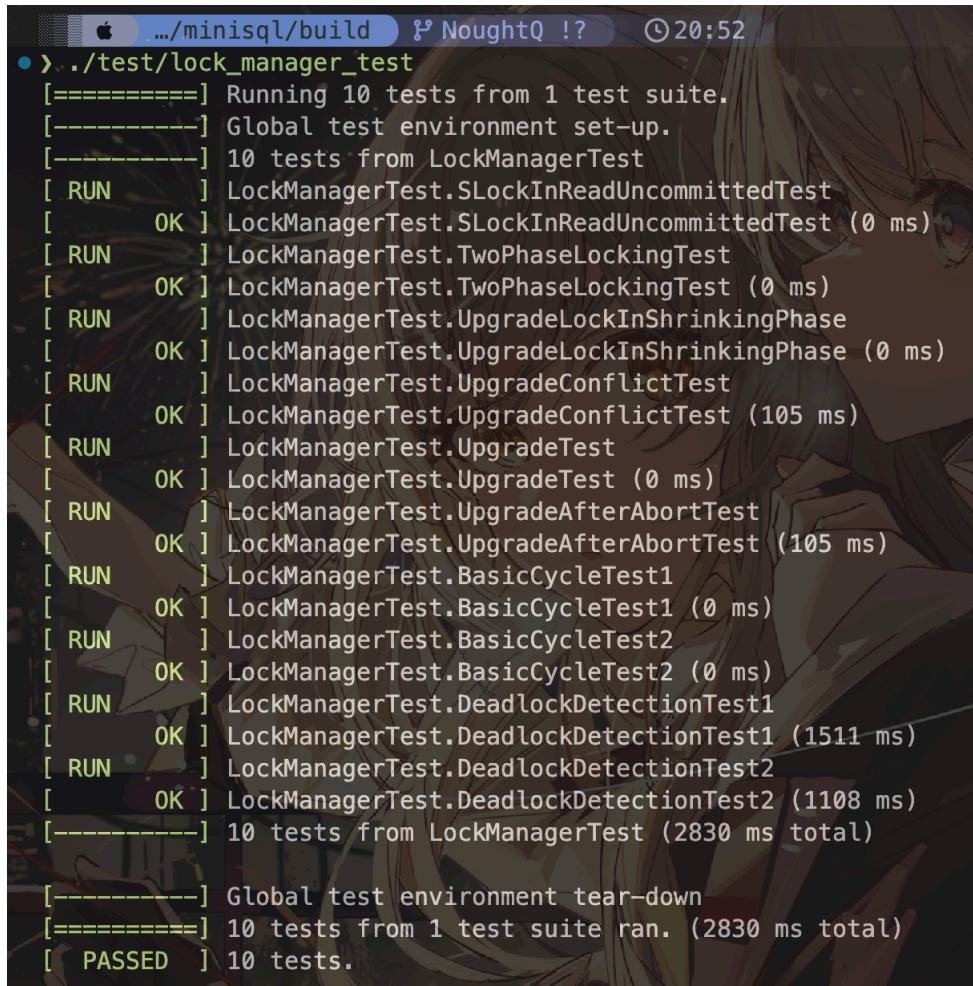
262             DeleteNode(newest_tid_in_cycle);
263         }
264
265         // wait for some intervals
266
266     std::this_thread::sleep_for(cycle_detection_interval_);
267 }
268 }
269
270 std::vector<std::pair<txn_id_t, txn_id_t>>
271 LockManager::GetEdgeList() {
272     std::vector<std::pair<txn_id_t, txn_id_t>> result;
273     for (const auto& elem : waits_for_) {
274         for (const auto& node : elem.second) {
275             result.push_back({elem.first, node});
276         }
277     }
278     return result;
279 }
```

2.7.3 测试

简要分析一下已有的测试代码：

- `SLockInReadUncommittedTest`: 检验在 READ_UNCOMMITTED 隔离级别下尝试获取共享锁的行为
- `TwoPhaseLockingTest`: 检验两阶段锁协议的基本功能，包括获取锁后进入 GROWING 阶段，以及释放锁后进入 SHRINKING 阶段，并在 SHRINKING 阶段尝试获取新锁（被拒）
- `UpgradeLockInShrinkingPhase`: 检验在事务的 SHRINKING 阶段尝试升级锁的行为（被拒）
- `UpgradeConflictTest`: 检验多个事务同时尝试升级锁时的冲突处理（后来者被中止）
- `UpgradeTest`: 检验单个事务的锁升级功能
- `UpgradeAfterAbortTest`: 检验事务被中止后的锁升级行为
- `BasicCycleTest1` 和 `BasicCycleTest2`: 检验死锁检测中的环检测功能
- `DeadlockDetectionTest1` 和 `DeadlockDetectionTest2`: 检验完整的死锁检测机制

测试结果如下：



```
.../minysql/build NoughtQ !? 20:52
● ./test/lock_manager_test
[=====] Running 10 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 10 tests from LockManagerTest
[ RUN   ] LockManagerTest.SLockInReadUncommittedTest
[ OK    ] LockManagerTest.SLockInReadUncommittedTest (0 ms)
[ RUN   ] LockManagerTest.TwoPhaseLockingTest
[ OK    ] LockManagerTest.TwoPhaseLockingTest (0 ms)
[ RUN   ] LockManagerTest.UpgradeLockInShrinkingPhase
[ OK    ] LockManagerTest.UpgradeLockInShrinkingPhase (0 ms)
[ RUN   ] LockManagerTest.UpgradeConflictTest
[ OK    ] LockManagerTest.UpgradeConflictTest (105 ms)
[ RUN   ] LockManagerTest.UpgradeTest
[ OK    ] LockManagerTest.UpgradeTest (0 ms)
[ RUN   ] LockManagerTest.UpgradeAfterAbortTest
[ OK    ] LockManagerTest.UpgradeAfterAbortTest (105 ms)
[ RUN   ] LockManagerTest.BasicCycleTest1
[ OK    ] LockManagerTest.BasicCycleTest1 (0 ms)
[ RUN   ] LockManagerTest.BasicCycleTest2
[ OK    ] LockManagerTest.BasicCycleTest2 (0 ms)
[ RUN   ] LockManagerTest.BasicCycleTest2
[ OK    ] LockManagerTest.BasicCycleTest2 (0 ms)
[ RUN   ] LockManagerTest.DeadlockDetectionTest1
[ OK    ] LockManagerTest.DeadlockDetectionTest1 (1511 ms)
[ RUN   ] LockManagerTest.DeadlockDetectionTest2
[ OK    ] LockManagerTest.DeadlockDetectionTest2 (1108 ms)
[-----] 10 tests from LockManagerTest (2830 ms total)

[-----] Global test environment tear-down
[-----] 10 tests from 1 test suite ran. (2830 ms total)
[ PASSED ] 10 tests.
```

Figure 26: 通过 lock_manager_test 的测试

可以看到，执行结果均符合预期，这证明了 Lock Manager 代码实现的正确性。并且我们认为已有的测试已经几乎完备，能够测试到绝大多数情况，故没有为该模块新增测试。

2.7.4 思考题

题目

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？

不独立 Lock Manager 模块，实际上就是重新设计索引等数据结构为基于锁的并发数据结构，并根据隔离等级决定每个操作施加锁的粒度。具体考虑到模块 3 中的 B+ 树的修改，我认为应该分设计思路与实现想法两部分处理。

2.7.4.1 设计思路

在框架中，`Page` 类已经完成了排他锁与共享锁的定义与实现，而 `B+` 树的节点页内容都是存储在 `Page` 类的 `data_` 成员中，所以我们没有必要再另外为 `B+` 树的节点重新定义与实现锁操作。

又由于 `B+` 树的任何操作都必须从根节点逐层向下直到叶节点，所以锁的粒度可以通过锁的节点的高度调节。比如如果我们在根节点所在页上加排他锁，那么实际上锁住了整棵 `B+` 树，其他事务无法读取根节点，从而无法执行任何操作。因此，我们不必再为不同锁的粒度要求设计其他的锁（比如 `table lock`）。

在模块 7 中一共要求实现 3 种隔离等级，分别是 `ReadUncommitted`、`ReadCommitted` 与 `RepeatableRead`，具体到读写操作，它们的要求如下：

- **ReadUncommitted**: 事务可以读取未提交的数据，因此读操作无需加锁，只有写操作需要加排他锁保护数据完整性
- **ReadCommitted**: 读操作需要加共享锁防止脏读，但读取完成后立即释放；写操作加排他锁并持有到事务提交
- **RepeatableRead**: 读操作加共享锁并持有到事务提交，确保同一事务中重复读取的一致性；写操作同样加排他锁持有到事务提交

根据它们的要求，我们可以进一步细化 `B+` 树三大操作的实现细节。

2.7.4.2 实现想法

- **查询操作**: 从根节点开始，沿查找路径对每个内部节点页加共享锁，当获得子节点的共享锁，释放父节点的锁。到达目标叶子节点后，根据隔离级别决定锁的持有时间：
 - `ReadUncommitted`: 直接读取不加锁；
 - `ReadCommitted`: 读完立即释放共享锁；
 - `RepeatableRead`: 持有共享锁到事务结束。
- **插入与删除操作**:

由于我们认为会引发分裂与合并的操作是少见的，大多数操作都不会引发 `B+` 树结构的改变，所以为了提高并发效率，我们可以采用“乐观锁”策略。

对于一般的操作，类似于查询操作，从根节点开始，沿路径对内部节点加共享锁，当获得子节点的锁，则释放父节点的锁。但到达叶子节点后，需要施加排他锁，如果不会发生分裂或合并，则正常根据隔离级别决定锁的持有时间：

- `ReadUncommitted`: 插入后立即释放叶子节点的排他锁；
- `ReadCommitted` 与 `RepeatableRead`: 持有排他锁到事务结束。

但如果会发生分裂或合并，则对父节点加排他锁，判断父节点插入新的记录后是否会发生分裂或合并。如果是，则接着递归地向上检查，直到某一父节点插入后不会引发分裂或合并（或者“蔓延”到根节点）。不过内部节点的排他锁的持有时间又根据隔离级别有一定差异：

- ReadUncommitted 与 ReadCommitted：完成结构变化后立即释放；
- RepeatableRead：持有直到事务结束。

但如果将上面的想法落实到代码中，那么我认为主要需要更改以下方法：

- FetchPage()：增加获取锁的可选参数；
- UnpinPage()：增加释放锁的可选操作。
- 根据上面的方法，对调用上面两个接口的方法作出对应修改（基本上对所有主要方法都需要修改，尤其是 FindLeafPage()、InsertIntoParent() 与 CoalesceOrRedistribute() 以及 GetValue() 与 Remove()）。

三、对代码框架的一些建议

- 像 select, insert, delete 和 update 这些操作在执行前都没有检查是否已选中数据库，建议添加这一检查逻辑。
- Executor 模块中删除数据库的函数 ExecuteEngine::ExecuteDropDatabase(*ast, *context) 有一块逻辑是把脏页写回马上就要删的文件中。我们认为这不太合理，建议为 DBStorageEngine 增加一个 bool 变量标记是否是销毁情况下的析构。如果是销毁情况，则所有的页都不需要刷新到磁盘。
- 建议为 minysql 程序加一个临时保存操作历史的功能，可以像 CLI 那样点击↑就能查看之前执行过的命令，这样就不用重复敲击 SQL 语句了。