

MiniSQL 小组总体报告

课程实验报告
数据库系统 (2024 年 春夏学期)
浙江大学

指导教师：苗晓晔

完成日期：2024 年 6 月 15 日

Made
With L^AT_EX

目录

第一部分 小组整体报告	4
第一章 分工	6
第二章 测试与结果	8
第二部分 个人详细报告	17
第三章 模块 1-3	19
3.1 模块 1	20
3.1.1 设计与实现	20
3.1.2 测试	21
3.2 模块 2	23
3.2.1 设计与实现	23
3.2.2 Bonus: 堆表加速 Row 的插入与查询	25
3.2.3 测试	27
3.3 模块 3	29
3.3.1 设计与实现	29
3.3.2 测试	32
第四章 模块 7	33
4.1 LockManager	33
4.2 测试	35
4.3 思考题	38
4.3.1 锁的粒度与隔离级别	38
4.3.2 具体的 B+ 树操作设计	39
第三部分 个人详细报告	41
第五章 模块 4-6	43
5.1 模块 4	43
5.1.1 模块概述	43
5.1.2 具体实现	44
5.1.3 测试	45
5.2 模块 5	48
5.2.1 模块概述	48
5.2.2 具体实现	50

5.2.3	测试	51
5.3	模块 6	54
5.3.1	模块概述	54
5.3.2	具体实现	54
5.3.3	测试	55
5.3.4	思考题	56
5.4	Bonus: Clock replacer	58
5.4.1	概述	58
5.4.2	实现	59
5.4.3	测试	60

第一部分

小组整体报告

第一章 分工

本次实验我们完成了模块 1-7，我们的任务分工如下表所示。

模块	分工
1	
2-3	
4-6	
7	

表 1.1: 任务分工

下面是使用命令 `git log --pretty='format:%as %h %cn: %s'` 导出的 Git Log，以证明我们的工作：

```
1 2024-06-14 0263bcb ***: update 6_test
2 2024-06-14 be5af93 ***: dlog remove
3 2024-06-14 ac17816 ***: [fix]: remove log
4 2024-06-14 79d30d3 ***: modify
5 2024-06-14 070451e ***: [fix]: Minor Update
6 2024-06-14 dcae6b1 ***: [fix](execute_engine): CreateTable Unique index should be
    created for each instead of all.
7 2024-06-14 ec8181c ***: [fix](catalog.cpp): Insert All Data When CreateIndex
8 2024-06-14 c383bad ***: [fix](test): minor update
9 2024-06-14 54982b4 ***: [fix](table_heap_test): Memory Leak.
10 2024-06-14 08f7410 ***: [perf](table_heap): faster insert algorithm.
11 2024-06-14 be3700e ***: [fix](table_iterator.h): Row Memory Leak.
12 2024-06-14 6128ff0 ***: [fix](recovery_test): Fix redefinition for static member
    accross different test unit.
13 2024-06-14 f3cd2f3 ***: [fix]: minor update.
14 2024-06-14 544e7b6 ***: [fix](buffer_pool_manager.cpp): Page Double Erase.
15 2024-06-14 33d8a23 ***: [fix](b_plus_tree_internal_page.cpp): Memcpy is not safe for
    overlapping dst and src. Use memmove.
16 2024-06-14 90e1034 ***: [fix](b_plus_tree_test.cpp): Now Free of Memory Leak.
17 2024-06-14 7769836 ***: [fix](b_plus_tree_index_test.cpp): Now Free of Memory Leak. 18 2024-06
-14 c1a4807 ***: [fix](b_plus_tree_index_test.cpp): Memory Leak and DB File
    reuse.
19 2024-06-14 f23641d ***: [fix](row.cpp): Memory Leak.
20 2024-06-13 4ee7a3d ***: Merge branch 'master' of git.zju.edu.cn:*/minisql 21 2024-06-13
3d104a4 ***: update test
22 2024-06-13 b2c4b69 ***: [fix](executor): supress output for execfile
23 2024-06-13 dbf74e8 ***: [fix](commands.sql): comment format
24 2024-06-13 4e0a104 ***: [feat](main.cpp): support redirection log
```

```

25 2024-06-13 53cd30d ***: [fix](main.cpp): end of file
26 2024-06-13 48dd8ce ***: 之前改错了 (
27 2024-06-13 e7d3aef ***: Update file execute_engine.cpp
28 2024-06-13 96413b9 ***: [feat](test): 1, 2 and 7 test
29 2024-06-13 e5b5d06 ***: [fix](lock_mamanger): minor update
30 2024-06-13 802a932 ***: [docs]: update
31 2024-06-13 bfa96de ***: Merge branch minisql:master into master
32 2024-06-13 3b08e20 ***: fix execute
33 2024-06-12 5d0b216 thebigdoudou: [bugfix]fixed some bugs.
34 2024-06-12 6885896 ***: A little modification
35 2024-06-12 114f4e0 ***: [feat](LockManager): All Tests Passed.
36 2024-06-10 9fa21d3 ***: [feat](BPlusTree_iter): BPlusTreeTests Passed
37 2024-06-10 8e8e2e9 ***: [feat](BPlusTree_iter): fin
38 2024-06-09 2a5030c ***: Merge branch minisql:master into master
39 2024-06-09 e3a91d5 ***: executor passed
40 2024-06-09 b3ffd41 thebigdoudou: [bugfix]fixed some bugs.
41 2024-06-09 9d6f713 thebigdoudou: [bugfix]fixed some bugs.
42 2024-06-07 70e4e03 ***: catalog update
43 2024-06-06 9ea752b ***: [feat](BPlusTree): BPlusTreeTests Passed (tested for 20 times)
44 2024-06-06 e5e96ca ***: [fix](disk_manager.cpp): double free warning
45 2024-06-06 718673c ***: catalog passed
46 2024-06-06 28bdfd0 ***: [feat](BPlusTree): Insert
47 2024-06-05 6f1ff1e thebigdoudou: [bugfix]fixed bug in IndexInfo::CreateIndex().
48 2024-06-04 315de74 ***: Replace recovery_manager.h    第六模块还有一个题目未描述的
    log_manager.h (不实现已通过测试)
49 2024-06-04 d365000 ***: Replace log_rec.h
50 2024-06-03 ca07aa0 ***: Update 2 files
51 2024-06-03 a1f39ea ***: Replace clock_replacer_test
52 2024-06-03 4c01793 ***: Update clock_replacer_test
53 2024-06-03 982958e ***: Add clock_replacer_test
54 2024-06-03 6e1d8b2 ***: Update clock_replacer.cpp
55 2024-06-03 8b6006f ***: Update clock_replacer.h
56 2024-06-03 247d52e ***: [feat](storage/table_heap): TableHeapTest Passed
57 2024-06-03 c9e6883 ***: [feat](src/record): TupleTest Passed
58 2024-06-03 53c6b67 ***: [refactor](buffer_pool_manager.cpp): optimize code structure 59 2024-06-03
924a7f9 ***: [perf](disk_manager.cpp): no need to write zero to disk when
    allocating pages
60 2024-06-03 f98c82e ***: [feat](buffer_pool_manager.cpp): BufferPoolManagerTest Passed 61 2024-06
-03 86798f8 ***: [feat](lru_replacer.cpp): LRURewriterTest Passed
62 2024-06-03 ac66ee3 ***: [feat](disk_manager.cpp): DiskManagerTest Passed
63 2024-06-02 1942544 ***: Merge branch minisql:master into master
64 2024-06-02 f279bd8 ***: [feat](bitmap_page.cpp): DiskManagerTest.BitMapPageTest Passed 65 2024-06
-02 33a25dc thebigdoudou: [bugfix]optimized some test cases and comments.
66 2024-06-02 aa96103 ***: Update bitmap_page.cpp

```

Listing 1.1: Git Log

第二章 测试与结果

我们的代码成功通过了框架提供的测试和我们自行设计的测试。关于每个模块单元测试的具体情况，请见个人详细报告。

```
1 ~/m/build (master)> test/minisql_test
2 [=====] Running 38 tests from 12 test suites.
3 [-----] Global test environment set-up.
4 [-----] 2 tests from BufferPoolManagerTest
5 [ RUN      ] BufferPoolManagerTest.BinaryDataTest
6 [          OK ] BufferPoolManagerTest.BinaryDataTest (0 ms)
7 [ RUN      ] BufferPoolManagerTest.WriteDataTest
8 [          OK ] BufferPoolManagerTest.WriteDataTest (0 ms)
9 [-----] 2 tests from BufferPoolManagerTest (1 ms total)
10
11 [-----] 1 test from ClockReplacerTest
12 [ RUN      ] ClockReplacerTest.SampleTest
13 [          OK ] ClockReplacerTest.SampleTest (0 ms)
14 [-----] 1 test from ClockReplacerTest (0 ms total)
15
16 [-----] 1 test from LRUReplacerTest
17 [ RUN      ] LRUReplacerTest.SampleTest
18 [          OK ] LRUReplacerTest.SampleTest (0 ms)
19 [-----] 1 test from LRUReplacerTest (0 ms total)
20
21 [-----] 4 tests from CatalogTest
22 [ RUN      ] CatalogTest.CatalogMetaTest
23 [          OK ] CatalogTest.CatalogMetaTest (0 ms)
24 [ RUN      ] CatalogTest.CatalogTableTest
25 [          OK ] CatalogTest.CatalogTableTest (55 ms)
26 [ RUN      ] CatalogTest.CatalogIndexTest
27 [          OK ] CatalogTest.CatalogIndexTest (30 ms)
28 [ RUN      ] CatalogTest.CatalogAllTest
29 [          OK ] CatalogTest.CatalogAllTest (29 ms)
30 [-----] 4 tests from CatalogTest (116 ms total)
31
32 [-----] 12 tests from LockManagerTest
33 [ RUN      ] LockManagerTest.SLockInReadUncommittedTest
34 [          OK ] LockManagerTest.SLockInReadUncommittedTest (0 ms)
35 [ RUN      ] LockManagerTest.TwoPhaseLockingTest
36 [          OK ] LockManagerTest.TwoPhaseLockingTest (0 ms)
37 [ RUN      ] LockManagerTest.UpgradeLockInShrinkingPhase
38 [          OK ] LockManagerTest.UpgradeLockInShrinkingPhase (0 ms)
39 [ RUN      ] LockManagerTest.UpgradeConflictTest
```

```

40 [      OK ] LockManagerTest.UpgradeConflictTest (100 ms)
41 [ RUN      ] LockManagerTest.UpgradeTest
42 [      OK ] LockManagerTest.UpgradeTest (0 ms)
43 [ RUN      ] LockManagerTest.UpgradeAfterAbortTest
44 [      OK ] LockManagerTest.UpgradeAfterAbortTest (100 ms)
45 [ RUN      ] LockManagerTest.BasicCycleTest1
46 [      OK ] LockManagerTest.BasicCycleTest1 (0 ms)
47 [ RUN      ] LockManagerTest.BasicCycleTest2
48 [      OK ] LockManagerTest.BasicCycleTest2 (0 ms)
49 [ RUN      ] LockManagerTest.DeadlockDetectionTest1
50 [      OK ] LockManagerTest.DeadlockDetectionTest1 (1500 ms)
51 [ RUN      ] LockManagerTest.DeadlockDetectionTest2
52 [      OK ] LockManagerTest.DeadlockDetectionTest2 (1101 ms)
53 [ RUN      ] LockManagerTest.BulkUpdateTest
54 [      OK ] LockManagerTest.BulkUpdateTest (1500 ms)
55 [ RUN      ] LockManagerTest.BulkTwoPhaseLockTest
56 [      OK ] LockManagerTest.BulkTwoPhaseLockTest (1105 ms)
57 [-----] 12 tests from LockManagerTest (5410 ms total)
58
59 [-----] 2 tests from TableHeapTest
60 [ RUN      ] TableHeapTest.TableIteratorTest
61 [      OK ] TableHeapTest.TableIteratorTest (254 ms)
62 [ RUN      ] TableHeapTest.TableHeapSampleTest
63 [      OK ] TableHeapTest.TableHeapSampleTest (223 ms)
64 [-----] 2 tests from TableHeapTest (477 ms total)
65
66 [-----] 3 tests from RecoveryManagerTest
67 [ RUN      ] RecoveryManagerTest.RedoTest
68 [      OK ] RecoveryManagerTest.RedoTest (0 ms)
69 [ RUN      ] RecoveryManagerTest.UndoTest
70 [      OK ] RecoveryManagerTest.UndoTest (0 ms)
71 [ RUN      ] RecoveryManagerTest.RecoveryTest
72 [      OK ] RecoveryManagerTest.RecoveryTest (0 ms)
73 [-----] 3 tests from RecoveryManagerTest (0 ms total)
74
75 [-----] 4 tests from ExecutorTest
76 [ RUN      ] ExecutorTest.SimpleSeqScanTest
77 [      OK ] ExecutorTest.SimpleSeqScanTest (33 ms)
78 [ RUN      ] ExecutorTest.SimpleDeleteTest
79 [      OK ] ExecutorTest.SimpleDeleteTest (144 ms)
80 [ RUN      ] ExecutorTest.SimpleRawInsertTest
81 [      OK ] ExecutorTest.SimpleRawInsertTest (32 ms)
82 [ RUN      ] ExecutorTest.SimpleUpdateTest
83 [      OK ] ExecutorTest.SimpleUpdateTest (39 ms)
84 [-----] 4 tests from ExecutorTest (249 ms total)
85
86 [-----] 4 tests from BPlusTreeTests
87 [ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
88 [      OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (0 ms)
89 [ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
90 [      OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (15 ms)
91 [ RUN      ] BPlusTreeTests.SampleTest

```



```

92 [          OK ] BPlusTreeTests.SampleTest (464 ms)
93 [ RUN          ] BPlusTreeTests.IndexIteratorTest
94 [          OK ] BPlusTreeTests.IndexIteratorTest (17 ms)
95 [-----] 4 tests from BPlusTreeTests (498 ms total)
96
97 [-----] 1 test from PageTests
98 [ RUN          ] PageTests.IndexRootsPageTest
99 [          OK ] PageTests.IndexRootsPageTest (0 ms)
100 [-----] 1 test from PageTests (0 ms total)
101
102 [-----] 2 tests from TupleTest
103 [ RUN          ] TupleTest.FieldSerializeDeserializeTest
104 [          OK ] TupleTest.FieldSerializeDeserializeTest (0 ms)
105 [ RUN          ] TupleTest.RowTest
106 [          OK ] TupleTest.RowTest (0 ms)
107 [-----] 2 tests from TupleTest (0 ms total)
108
109 [-----] 2 tests from DiskManagerTest
110 [ RUN          ] DiskManagerTest.BitMapPageTest
111 [          OK ] DiskManagerTest.BitMapPageTest (2 ms)
112 [ RUN          ] DiskManagerTest.FreePageAllocationTest
113 [          OK ] DiskManagerTest.FreePageAllocationTest (108 ms)
114 [-----] 2 tests from DiskManagerTest (110 ms total)
115
116 [-----] Global test environment tear-down
117 [=====] 38 tests from 12 test suites ran. (6865 ms total)
118 [ PASSED   ] 38 tests.

```

Listing 2.1: 单元测试结果

在验收时，我们已经展示过小数据集和大数据集的情况。大数据集输出较多，不展示，下面展示小数据集的情况。

```

1 minisql > show databases;
2 Empty set (0.00 sec)
3 minisql > create database db0;
4 minisql > create database db1;
5 minisql > create database db2;
6 minisql > show databases;
7 +-----+
8 | Database |
9 +-----+
10 | db2      |
11 | db1      |
12 | db0      |
13 +-----+
14 minisql > use db0;
15 Database changed
16 minisql > show tables;
17 +-----+
18 | Tables_in_db0 |
19 +-----+
20 +-----+

```

```

21 minisql > create table account(
22     id int,
23     name char(16) unique,
24     balance float,
25     primary key(id)
26 );
27 minisql > show tables;
28 +-----+
29 | Tables_in_db0 |
30 +-----+
31 | account      |
32 +-----+
33 minisql > execfile "/home/bowling233/minisql/sql_gen/account00.txt";
34 Execfile started, output suppressed.
35 Execfile finished in 5435 ms
36 minisql > select * from account where id = 12500000;
37 +-----+-----+-----+
38 | id      | name      | balance  |
39 +-----+-----+-----+
40 | 12500000 | name00000 | 514.349976 |
41 +-----+-----+-----+
42 1 row in set(0.0000 sec).
43 minisql > select * from account where id = 12509999;
44 +-----+-----+-----+
45 | id      | name      | balance  |
46 +-----+-----+-----+
47 | 12509999 | name09999 | 86.269997 |
48 +-----+-----+-----+
49 1 row in set(0.0000 sec).
50 minisql > select * from account where id = 12510000;
51 Empty set(0.0000 sec).
52 minisql > select * from account where name = "name00000";
53 +-----+-----+-----+
54 | id      | name      | balance  |
55 +-----+-----+-----+
56 | 12500000 | name00000 | 514.349976 |
57 +-----+-----+-----+
58 1 row in set(0.0000 sec).
59 minisql > select * from account where name = "name09999";
60 +-----+-----+-----+
61 | id      | name      | balance  |
62 +-----+-----+-----+
63 | 12509999 | name09999 | 86.269997 |
64 +-----+-----+-----+
65 1 row in set(0.0000 sec).
66 minisql > select * from account where name = "name10000";
67 Empty set(0.0000 sec).
68 minisql > select * from account where id < 12500200 and name < "name00005";
69 +-----+-----+-----+
70 | id      | name      | balance  |
71 +-----+-----+-----+
72 | 12500000 | name00000 | 514.349976 |

```

```

73 | 12500001 | name00001 | 103.139999 |
74 | 12500002 | name00002 | 981.859985 |
75 | 12500003 | name00003 | 926.510010 |
76 | 12500004 | name00004 | 4.870000 |
77 +-----+-----+-----+
78 5 row in set(0.0000 sec).
79 minisql > select * from account where name = "name00001";
80 +-----+-----+-----+
81 | id      | name      | balance  |
82 +-----+-----+-----+
83 | 12500001 | name00001 | 103.139999 |
84 +-----+-----+-----+
85 1 row in set(0.0000 sec).
86 minisql > delete from account where name = "name00001";
87 Query OK, 1 row affected(0.1900 sec).
88 minisql > select * from account where name = "name00001";
89 Empty set(0.0000 sec).
90 minisql > insert into account values(1, "name00001", 0);
91 Query OK, 1 row affected(0.0000 sec).
92 minisql > select * from account where name = "name00001";
93 +----+-----+-----+
94 | id | name      | balance |
95 +----+-----+-----+
96 | 1  | name00001 | 0.000000 |
97 +----+-----+-----+
98 1 row in set(0.0000 sec).
99 minisql > delete from account where name = "name00001";
100 Query OK, 1 row affected(0.1780 sec).
101 minisql > insert into account values(12500001, "name00001", 103.14);
102 Query OK, 1 row affected(0.0010 sec).
103 minisql > create index idx01 on account(name);
104 minisql > select * from account where id = 12500000;
105 +-----+-----+-----+
106 | id      | name      | balance  |
107 +-----+-----+-----+
108 | 12500000 | name00000 | 514.349976 |
109 +-----+-----+-----+
110 1 row in set(0.0000 sec).
111 minisql > select * from account where id = 12509999;
112 +-----+-----+-----+
113 | id      | name      | balance  |
114 +-----+-----+-----+
115 | 12509999 | name09999 | 86.269997 |
116 +-----+-----+-----+
117 1 row in set(0.0000 sec).
118 minisql > select * from account where id = 12510000;
119 Empty set(0.0000 sec).
120 minisql > select * from account where name = "name00000";
121 +-----+-----+-----+
122 | id      | name      | balance  |
123 +-----+-----+-----+
124 | 12500000 | name00000 | 514.349976 |

```

```

125 +-----+-----+-----+
126 1 row in set(0.0000 sec).
127 minisql > select * from account where name = "name09999";
128 +-----+-----+-----+
129 | id      | name      | balance  |
130 +-----+-----+-----+
131 | 12509999 | name09999 | 86.269997 |
132 +-----+-----+-----+
133 1 row in set(0.0000 sec).
134 minisql > select * from account where name = "name10000";
135 Empty set(0.0000 sec).
136 minisql > select * from account where id < 12500200 and name < "name00005";
137 +-----+-----+-----+
138 | id      | name      | balance  |
139 +-----+-----+-----+
140 | 12500000 | name00000 | 514.349976 |
141 | 12500002 | name00002 | 981.859985 |
142 | 12500003 | name00003 | 926.510010 |
143 | 12500004 | name00004 | 4.870000  |
144 | 12500001 | name00001 | 103.139999 |
145 +-----+-----+-----+
146 5 row in set(0.0000 sec).
147 minisql > select * from account where name = "name00001";
148 +-----+-----+-----+
149 | id      | name      | balance  |
150 +-----+-----+-----+
151 | 12500001 | name00001 | 103.139999 |
152 +-----+-----+-----+
153 1 row in set(0.0000 sec).
154 minisql > delete from account where name = "name00001";
155 Query OK, 1 row affected(0.2570 sec).
156 minisql > select * from account where name = "name00001";
157 Empty set(0.0000 sec).
158 minisql > insert into account values(1, "name00001", 0);
159 Query OK, 1 row affected(0.0000 sec).
160 minisql > select * from account where name = "name00001";
161 +----+-----+-----+
162 | id | name      | balance  |
163 +----+-----+-----+
164 | 1  | name00001 | 0.000000 |
165 +----+-----+-----+
166 1 row in set(0.0000 sec).
167 minisql > delete from account where name = "name00001";
168 Query OK, 1 row affected(0.2580 sec).
169 minisql > insert into account values(12500001, "name00001", 103.14);
170 Query OK, 1 row affected(0.0000 sec).
171 minisql > drop index idx01;
172 minisql > select * from account where id = 12500000;
173 +-----+-----+-----+
174 | id      | name      | balance  |
175 +-----+-----+-----+
176 | 12500000 | name00000 | 514.349976 |

```

```

177 +-----+-----+-----+
178 1 row in set(0.2540 sec).
179 minisql > select * from account where id = 12509999;
180 +-----+-----+-----+
181 | id      | name      | balance   |
182 +-----+-----+-----+
183 | 12509999 | name09999 | 86.269997 |
184 +-----+-----+-----+
185 1 row in set(0.2440 sec).
186 minisql > select * from account where id = 12510000;
187 Empty set(0.2410 sec).
188 minisql > select * from account where name = "name00000";
189 +-----+-----+-----+
190 | id      | name      | balance   |
191 +-----+-----+-----+
192 | 12500000 | name00000 | 514.349976 |
193 +-----+-----+-----+
194 1 row in set(0.2390 sec).
195 minisql > select * from account where name = "name09999";
196 +-----+-----+-----+
197 | id      | name      | balance   |
198 +-----+-----+-----+
199 | 12509999 | name09999 | 86.269997 |
200 +-----+-----+-----+
201 1 row in set(0.2450 sec).
202 minisql > select * from account where name = "name10000";
203 Empty set(0.2530 sec).
204 minisql > select * from account where id < 12500200 and name < "name00005";
205 +-----+-----+-----+
206 | id      | name      | balance   |
207 +-----+-----+-----+
208 | 12500000 | name00000 | 514.349976 |
209 | 12500002 | name00002 | 981.859985 |
210 | 12500003 | name00003 | 926.510010 |
211 | 12500004 | name00004 | 4.870000   |
212 | 12500001 | name00001 | 103.139999 |
213 +-----+-----+-----+
214 5 row in set(0.2440 sec).
215 minisql > select * from account where name = "name00001";
216 +-----+-----+-----+
217 | id      | name      | balance   |
218 +-----+-----+-----+
219 | 12500001 | name00001 | 103.139999 |
220 +-----+-----+-----+
221 1 row in set(0.2460 sec).
222 minisql > delete from account where name = "name00001";
223 Query OK, 1 row affected(0.2450 sec).
224 minisql > select * from account where name = "name00001";
225 Empty set(0.2440 sec).
226 minisql > insert into account values(1, "name00001", 0);
227 Query OK, 1 row affected(0.0000 sec).
228 minisql > select * from account where name = "name00001";

```

```

229 +-----+-----+-----+
230 | id | name      | balance |
231 +-----+-----+-----+
232 | 1  | name00001 | 0.000000 |
233 +-----+-----+-----+
234 1 row in set(0.2600 sec).
235 minisql > delete from account where name = "name00001";
236 Query OK, 1 row affected(0.2630 sec).
237 minisql > insert into account values(12500001, "name00001", 103.14);
238 Query OK, 1 row affected(0.0000 sec).
239 minisql > select * from account where name = "name45678";
240 Empty set(0.2570 sec).
241 minisql > drop table account;
242 minisql > show tables;
243 +-----+
244 | Tables_in_db0 |
245 +-----+
246 +-----+
247 minisql >
248 minisql > EOF Received, exit.

```

Listing 2.2: SQL 测试结果

可以看到各项功能执行正常。在现场验收时，我们也顺利展示完成了所有功能。

第二部分

个人详细报告

第三章 模块 1-3

模块 1-3 涉及的代码比较底层（如比特、字节级别的数据操作），各项数据都需要仔细计算。在阅读 1-3 模块的代码时，我绘制了一张图辅助自己理解，也基本涵盖了我对这几个模块的理解。图中主要展示了这些基础数据结构之间的组合关系：

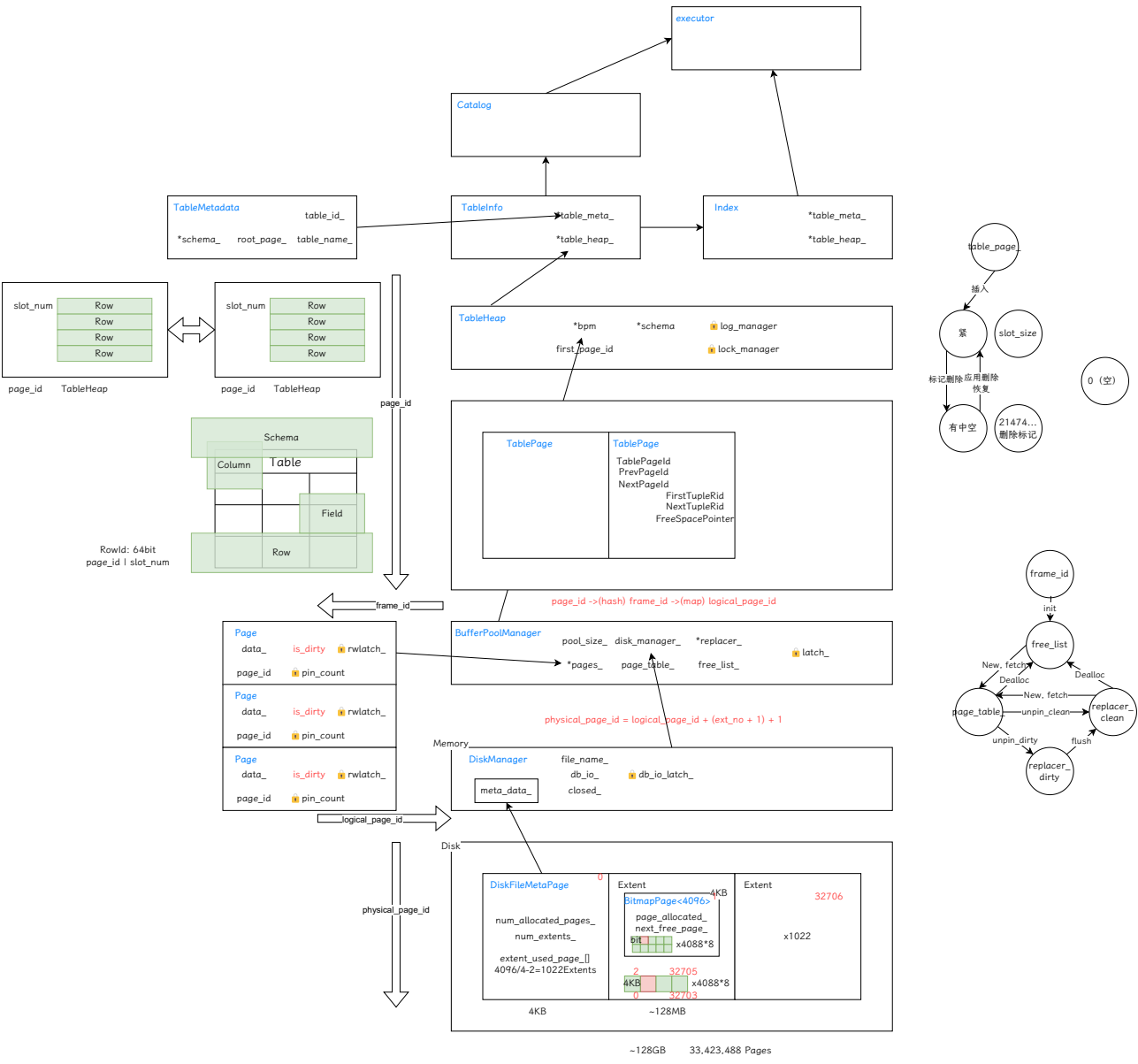


图 3.1: 基础数据结构之间的关系

3.1 模块 1

3.1.1 设计与实现

模块 1 比较简单，没有什么可以自己设计的余地，就是按照给定的要求实现即可。值得一提的是 Bitmap 的分配方式，最 naive 的实现是每次分配都从头开始遍历，寻找空闲的页。但我通过合理利用 `next_free_page_`，可以使得，只要调用者总是倾向于释放比编号较大的页（我认为这是比较普遍的情况），那么摊还复杂度为 $O(1)$ 。下面的两个代码清单展示了 Bitmap 的分配和回收实现：

```
1 template <size_t PageSize>
2 bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset) {
3     // Check if free page left
4     if (next_free_page_ >= GetMaxSupportedSize()) {
5         return false;
6     }
7     uint32_t byte_index = next_free_page_ / 8;
8     uint8_t bit_index = next_free_page_ % 8;
9     // Allocate next free page
10    bytes[byte_index] |= (1 << bit_index);
11    page_offset = next_free_page_;
12    // Find next free page
13    page_allocated_++;
14    if (page_allocated_ == GetMaxSupportedSize()) {
15        next_free_page_ = GetMaxSupportedSize();
16        return true;
17    }
18    do {
19        next_free_page_ = (next_free_page_ + 1) % GetMaxSupportedSize();
20        if (IsPageFree(next_free_page_)) {
21            return true;
22        }
23    } while (next_free_page_ != page_offset);
24    throw std::exception();
25 }
```

Listing 3.1: Bitmap 分配

```
1 template <size_t PageSize>
2 bool BitmapPage<PageSize>::DeAllocatePage(uint32_t page_offset) {
3     if (page_offset >= GetMaxSupportedSize()) {
4         throw std::out_of_range("Page offset is out of range");
5     }
6     // Check if page is already free
7     if (IsPageFree(page_offset)) {
8         return false;
9     }
10    // De-allocate page
11    uint32_t byte_index = page_offset / 8;
12    uint8_t bit_index = page_offset % 8;
13    bytes[byte_index] &= ~(1 << bit_index);
14    // Update next_free_page_
15    if (page_offset < next_free_page_) {
16        next_free_page_ = page_offset;
17    }
18 }
```

```

17 }
18 page_allocated--;
19 return true;
20 }

```

Listing 3.2: Bitmap 回收

3.1.2 测试

本模块包含几个测试单元：

- `disk_manager_test`：测试 `BitmapPage` 的分配和回收。
- `lru_replacer_test`：测试 `LRUReplacer` 的基本功能。
- `buffer_pool_manager_test`：测试 `BufferPoolManager` 的基本功能。

实话说这几个测试单元已经测试得非常详尽了。比如 `BitmapPage` 的测试，测试了分配和回收的基本功能，以及分配和回收的边界情况。在本模块，我只为 `buffer_pool_manager` 补充了一个测试 `WriteDataTest`，测试了写入和读取的基本功能。基本步骤如下：

- 创建一个 `BufferPoolManager` 实例。
- 从 `BufferPoolManager` 中获取一个页，写入数据。
- `Unpin` 这个页，并写入磁盘。
- 再获取这个页，读取数据。应当与写入的数据一致。
- 再次修改这个页，写入数据。
- 再次获取这个页，应当读取到修改后的数据。
- `Unpin` 这个页，写入磁盘。
- 重复上述步骤。
- 释放 `BufferPoolManager`。

下面的代码清单展示了 `WriteDataTest` 的实现：

```

1 TEST(BufferPoolManagerTest, WriteDataTest) {
2     const std::string db_name = "bpm_test.db";
3     const size_t buffer_pool_size = 10;
4     std::random_device r;
5     std::default_random_engine rng(r());
6     std::uniform_int_distribution<unsigned> uniform_dist(0, 127);
7     remove(db_name.c_str());
8     auto *disk_manager = new DiskManager(db_name);
9     auto *bpm = new BufferPoolManager(buffer_pool_size, disk_manager);
10    page_id_t page_id_temp;
11    auto *page0 = bpm->NewPage(page_id_temp);
12
13    ASSERT_NE(nullptr, page0);
14    EXPECT_EQ(0, page_id_temp);
15

```

```

16  char random_binary_data[PAGE_SIZE];
17  for (char &i : random_binary_data) {
18      i = uniform_dist(rng);
19  }
20
21  random_binary_data[PAGE_SIZE / 2] = '\0';
22  random_binary_data[PAGE_SIZE - 1] = '\0';
23
24  std::memcpy(page0->GetData(), random_binary_data, PAGE_SIZE);
25  EXPECT_EQ(0, std::memcmp(page0->GetData(), random_binary_data, PAGE_SIZE));
26
27  EXPECT_TRUE(bpm->UnpinPage(0, true));
28  EXPECT_TRUE(bpm->FlushPage(0));
29
30  auto *page1 = bpm->FetchPage(0);
31  EXPECT_NE(nullptr, page1);
32  for (int i = 0; i < PAGE_SIZE; i++) {
33      EXPECT_EQ(random_binary_data[i], page1->GetData()[i]);
34  }
35
36  for (char &i : random_binary_data) {
37      i = uniform_dist(rng);
38  }
39  std::memcpy(page1->GetData(), random_binary_data, PAGE_SIZE);
40
41  EXPECT_TRUE(bpm->UnpinPage(0, true));
42  EXPECT_TRUE(bpm->FlushPage(0));
43
44  auto *page2 = bpm->FetchPage(0);
45  EXPECT_NE(nullptr, page2);
46  for (int i = 0; i < PAGE_SIZE; i++) {
47      EXPECT_EQ(random_binary_data[i], page2->GetData()[i]);
48  }
49
50  disk_manager->Close();
51  remove(db_name.c_str());
52
53  delete bpm;
54  delete disk_manager;
55 }

```

Listing 3.3: BufferPoolManager 测试

下图展示了框架给定的测试和自行设计的测试的通过情况：

```
tmux /home/bowling233
bowling233@bowling-desktop ~/m/build (master)> test/1_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from BufferPoolManagerTest
[ RUN    ] BufferPoolManagerTest.WriteDataTest
[ OK     ] BufferPoolManagerTest.WriteDataTest (1 ms)
[-----] 1 test from BufferPoolManagerTest (1 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (1 ms total)
[ PASSED ] 1 test.
bowling233@bowling-desktop ~/m/build (master)> test/buffer_pool_manager_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from BufferPoolManagerTest
[ RUN    ] BufferPoolManagerTest.BinaryDataTest
[ OK     ] BufferPoolManagerTest.BinaryDataTest (0 ms)
[-----] 1 test from BufferPoolManagerTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 1 test.
bowling233@bowling-desktop ~/m/build (master)>
bowling233@bowling-desktop ~/m/build (master)> test/disk_manager_test
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from DiskManagerTest
[ RUN    ] DiskManagerTest.BitMapPageTest
[ OK     ] DiskManagerTest.BitMapPageTest (2 ms)
[ RUN    ] DiskManagerTest.FreePageAllocationTest
[ OK     ] DiskManagerTest.FreePageAllocationTest (124 ms)
[-----] 2 tests from DiskManagerTest (127 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (127 ms total)
[ PASSED ] 2 tests.
bowling233@bowling-desktop ~/m/build (master)> test/lru_replacer_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from LRUReplacerTest
[ RUN    ] LRUReplacerTest.SampleTest
[ 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.
bowling233@bowling-desktop ~/m/build (master)>
[0] 0:fish* "fish /home/bowling233" 09:01 13-Jun-24
```

图 3.2: 模块 1 测试通过情况

3.2 模块 2

3.2.1 设计与实现

本模块也很简单，按照要求实现即可。实验指导中提到的使用位图方式标记 null 的 Field 实现如下：

```
1 uint32_t Row::SerializeTo(char *buf, Schema *schema) const {
2     ASSERT(schema != nullptr, "Invalid schema before serialize.");
3     ASSERT(schema->GetColumnCount() == fields_.size(), "Fields size do not match schema's
4         column size.");
5     uint32_t offset = 0;
6     // rid
7     memcpy(buf + offset, &rid_, sizeof(RowId));
8     offset += sizeof(RowId);
9     // null bitmap
10    uint32_t null_size = (fields_.size() + 7) / 8;
11    char null_bitmap[null_size];
12    memset(null_bitmap, 0, null_size);
13    for (uint32_t i = 0; i < fields_.size(); i++) {
14        if (fields_[i]->IsNull()) {
15            null_bitmap[i / 8] |= (1 << (i % 8));
16        }
17    }
18    memcpy(buf + offset, null_bitmap, null_size);
19    offset += null_size;
20    // fields
21    for (uint32_t i = 0; i < fields_.size(); i++) {
22        offset += fields_[i]->SerializeTo(buf + offset);
23    }
24    return offset;
25 }
```

Listing 3.4: Field 实现

TablePage 已经完整提供，让我们阅读并总结一下要点：

- 插入：获取长度，找空 slot（疑问：为什么不直接从尾部呢？需要根据删除的方法而定），写入，调整长度、指针、tuple 记录数。
- 标记删除：操作 tuplesize（根据 slot 找），值得一提的用的似乎是异或操作，应该是因为这个运算可逆的性质，有助于之后的恢复。
- 更新：长度不够直接返回 false 无操作。否则，操作：读取旧长度，移动数据，写入新数据，更新所有 slot 的偏移。
- 删除：标记删除，调整指针、长度、记录数。
- IsDeleted 有两种情况：tuple_size 为 0（暂时没想到为什么会有这种情况）和 DELETEMASK。在 tablePage 各个函数中用于检测是否被删除，可以看出只要标记就认为是删除了。
- 回滚删除：因为咱不实现恢复机制所以无需关心。

根据 TablePage，我们可以设计 TableHeap 几个函数的实现：

- GetTuple：简单。
- InsertTuple：
 - 获取 row 的长度
 - 循环找到能放下的 page，否则新建
 - 存放 row
- UpdateTuple: RowId 允许被更改
 - 先尝试直接更新
 - 先 markdelete
 - 再调用插入

下面的代码清单展示了 TableHeap 插入函数的实现：

```

1 bool TableHeap::InsertTuple(Row &row, Txn *txn) {
2     auto row_size = row.GetSerializedSize(schema_);
3     if (row_size >= PAGE_SIZE) {
4         return false;
5     }
6     // Find the page which can save the tuple.
7     page_id_t pre_page_id, page_id;
8     page_id = pre_page_id = first_page_id_;
9     while (page_id != INVALID_PAGE_ID) {
10        auto page = reinterpret_cast<TablePage *>(buffer_pool_manager_>FetchPage(page_id));
11        if (page == nullptr) {
12            return false;
13        }
14        // try to insert the tuple into the page.

```

```

15     page->WLatch();
16     auto inserted_ = page->InsertTuple(row, schema_, txn, lock_manager_, log_manager_);
17     page->WUnlatch();
18     buffer_pool_manager_->UnpinPage(page_id, inserted_);
19     if (inserted_) {
20         return true;
21     }
22     // try next page
23     pre_page_id = page_id;
24     page_id = page->GetNextPageId();
25 }
26 // Create a new page
27 auto new_page = reinterpret_cast<TablePage *>(buffer_pool_manager_->NewPage(page_id));
28 if (new_page == nullptr) {
29     return false;
30 }
31 // Link the new page to the previous page.
32 if (pre_page_id != INVALID_PAGE_ID) {
33     auto pre_page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(
34         pre_page_id));
35     pre_page->WLatch();
36     pre_page->SetNextPageId(new_page->GetPageId());
37     pre_page->WUnlatch();
38     buffer_pool_manager_->UnpinPage(pre_page_id, true);
39 }
40 // Insert the tuple
41 new_page->WLatch();
42 new_page->Init(new_page->GetPageId(), pre_page_id, log_manager_, txn); // prev linked
43                                 here
44 if (!new_page->InsertTuple(row, schema_, txn, lock_manager_, log_manager_)) {
45     return false;
46 }
47 new_page->WUnlatch();
48 buffer_pool_manager_->UnpinPage(new_page->GetPageId(), true);
49 return true;
50 }

```

Listing 3.5: TableHeap 插入

TableIterator 的实现非常简单，递增时超过当前页的记录数就换页，直到找到一个未删除的记录。值得一提的是，TableIterator 框架给的构造函数是 explicit 的，但拷贝构造通常不应该是。应当删去。具体代码不在此处展示。

3.2.2 Bonus: 堆表加速 Row 的插入与查询

Bonus 要求优化堆表 (TableHeap) 以及数据页 (TablePage) 的实现，通过使用额外的空间记录一些元信息来加速 Row 的插入、查找和删除操作。在我看来，最耗时的是 Row 的插入操作，而查找和删除可以通过 RowId 准确定位到 PageId 和 SlotNum，我认为不存在进一步优化的空间。

插入操作朴素的实现是遍历当前堆表中所有数据页，找到空闲空间能够容纳 Row 的页表进行插入。容易看出这种插入方式是 $O(n)$ 级别的。因此，应当使用高效的数据结构记录每个页面的剩余空间，并提供高效的查找方式。首选的数据结构应该是优先队列等，但 STL 库中的优先队列无法遍历其中的元素，因

此我们退而求其次使用哈希表，即 `unordered_map` 记录每个 `TablePage` 的空闲空间，使用 `<algorithm>` 中的 `std::find_if` 进行查找。下面的代码清单展示了优化后的堆表插入实现：

```
1 bool TableHeap::InsertTuple(Row &row, Txn *txn) {
2     auto row_size = row.GetSerializedSize(schema_);
3     //DLOG(INFO) << "Row size: " << row_size;
4     if (row_size >= PAGE_SIZE) {
5         //DLOG(ERROR) << "The tuple is too large to insert.";
6         return false;
7     }
8     // Find the page which can save the tuple.
9     auto page = std::find_if(page_free_space_.begin(), page_free_space_.end(),
10                             [row_size](const auto &pair) { return pair.second >= row_size
11                                 + TablePage::SIZE_TUPLE; });
12     TablePage *page_to_insert = nullptr;
13     if (page == page_free_space_.end()) {
14         //DLOG(INFO) << "[InsertTuple] Create a new page.";
15         // Create a new page.
16         page_id_t new_page_id;
17         auto new_page = reinterpret_cast<TablePage *>(buffer_pool_manager_>NewPage(
18             new_page_id));
19         if (new_page == nullptr) return false;
20         // setup the new page
21         new_page->Init(new_page_id, page_free_space_.rbegin()->first, log_manager_, txn);
22         // link to the previous page
23         auto pre_page_id = page_free_space_.rbegin()->first;
24         auto pre_page = reinterpret_cast<TablePage *>(buffer_pool_manager_>FetchPage(
25             pre_page_id));
26         pre_page->WLatch();
27         pre_page->SetNextPageId(new_page_id);
28         pre_page->WUnlatch();
29         buffer_pool_manager_>UnpinPage(pre_page_id, true);
30         page_to_insert = new_page;
31     } else {
32         //DLOG(INFO) << "[InsertTuple] Insert into existing page.";
33         page_to_insert = reinterpret_cast<TablePage *>(buffer_pool_manager_>FetchPage(page->
34             first));
35     }
36     // Insert the tuple
37     //DLOG(INFO) << "Insert tuple into page " << page_to_insert->GetTablePageId();
38     page_to_insert->WLatch();
39     if (!page_to_insert->InsertTuple(row, schema_, txn, lock_manager_, log_manager_)) {
40         //DLOG(ERROR) << "InsertTuple Failed";
41         return false;
42     }
43     page_to_insert->WUnlatch();
44     // Update the free space of the page.
45     page_free_space_[page_to_insert->GetTablePageId()] = page_to_insert->
46         GetFreeSpaceRemaining();
47     //DLOG(INFO) << "Free space remaining: " << page_to_insert->GetFreeSpaceRemaining();
48     buffer_pool_manager_>UnpinPage(page_to_insert->GetPageId(), true);
49     return true;
50 }
```

45 }

Listing 3.6: TableHeap 插入（优化后）

这个记录表不仅要在插入时进行维护，还要在构造时维护，因此 TableHeap 的构造函数也需要修改：

```
1 explicit TableHeap(BufferPoolManager *buffer_pool_manager, page_id_t first_page_id,
    Schema *schema,
2     first_page_id_(first_page_id),
3     schema_(schema),
4     log_manager_(log_manager),
5     lock_manager_(lock_manager) {}
6     lock_manager_(lock_manager) {
7     auto page = reinterpret_cast<TablePage *>(buffer_pool_manager->FetchPage(
    first_page_id_));
8     assert(page != nullptr);
9     page_free_space_[first_page_id_] = page->GetFreeSpaceRemaining();
10    auto next_page_id = page->GetNextPageId();
11    buffer_pool_manager->UnpinPage(first_page_id_, false);
12    // fill page_free_space_
13    while (next_page_id != INVALID_PAGE_ID) {
14        page = reinterpret_cast<TablePage *>(buffer_pool_manager->FetchPage(next_page_id))
15        ;
16        assert(page != nullptr);
17        page_free_space_[next_page_id] = page->GetFreeSpaceRemaining();
18        next_page_id = page->GetNextPageId();
19        buffer_pool_manager->UnpinPage(next_page_id, false);
20    }
21 }
```

Listing 3.7: TableHeap 构造函数

这一优化其实是在完成整个实验主体后再进行的。模块单元测试的数据量太小，没有让我们注意到这一优化的必要性，而在执行验收流程中的插入步骤时，我们发现插入一万条数据就需要 35 秒，再插入一万条数据的时间竟然达到了 350 秒，这是不可接受的。通过对 Executor 的调用栈分析，我们发现热点位于堆表，才明白底层数据结构优化的重要之处。进行优化后，我们插入一万条数据的时间减少到了 5 秒，此后再插入几万条数据，每次只会有 1 秒左右的增长，不会再是平方级别，这验证了我们优化的有效性。同时，我们了解到有的组插入所有数据甚至需要十几分钟，这也表明我们在模块 1-2 中对底层数据结构的实现是比较高效的。

3.2.3 测试

本模块包含几个测试单元：

- tuple_test：测试 Tuple 的基本功能。
- table_heap_test：测试 TableHeap 的基本功能。

测试单元也已经相当完善：填充的数据包含了 Nullable 的情况，也进行了完整的检测。我只做了一些小的修改，补上了 TableIterator 的测试：

```
1 TEST(TableHeapTest, TableIteratorTest) {
2     // init testing instance
3     remove(db_file_name.c_str());
```



```

4  auto disk_mgr_ = new DiskManager(db_file_name);
5  auto bpm_ = new BufferPoolManager(DEFAULT_BUFFER_POOL_SIZE, disk_mgr_);
6  const int row_nums = 10000;
7  // create schema
8  std::vector<Column *> columns = {new Column("id", TypeId::kTypeInt, 0, false, false),
9                                   new Column("name", TypeId::kTypeChar, 64, 1, true,
10                                              false),
11                                   new Column("account", TypeId::kTypeFloat, 2, true,
12                                              false)};
13 auto schema = std::make_shared<Schema>(columns);
14 // 从 unordered_map 修改为 map 以提供顺序迭代
15 std::map<int64_t, Fields *> row_values;
16 uint32_t size = 0;
17 TableHeap *table_heap = TableHeap::Create(bpm_, schema.get(), nullptr, nullptr, nullptr);
18
19 for (int i = 0; i < row_nums; i++) {
20     int32_t len = RandomUtils::RandomInt(0, 64);
21     char *characters = new char[len];
22     RandomUtils::RandomString(characters, len);
23     Fields *fields =
24         new Fields{Field(TypeId::kTypeInt, i), Field(TypeId::kTypeChar, const_cast<char
25                                                         *>(characters), len, true),
26                    Field(TypeId::kTypeFloat, RandomUtils::RandomFloat(-999.f, 999.f))};
27     Row row(*fields);
28     ASSERT_TRUE(table_heap->InsertTuple(row, nullptr));
29     if (row_values.find(row.GetRowId().Get()) != row_values.end()) {
30         std::cout << row.GetRowId().Get() << std::endl;
31         ASSERT_TRUE(false);
32     } else {
33         row_values.emplace(row.GetRowId().Get(), fields);
34         size++;
35     }
36     delete[] characters;
37 }
38
39 ASSERT_EQ(row_nums, row_values.size());
40 ASSERT_EQ(row_nums, size);
41 auto itr = table_heap->Begin(nullptr);
42 for (auto row_kv : row_values) {
43     size--;
44     Row row = *itr;
45     itr++;
46     // 迭代器获得的 Row 的 Id 应当与顺序相同。
47     ASSERT_EQ(row.GetRowId().Get(), row_kv.first);
48     ASSERT_EQ(schema.get()->GetColumnCount(), row.GetFields().size());
49     for (size_t j = 0; j < schema.get()->GetColumnCount(); j++) {
50         ASSERT_EQ(CmpBool::kTrue, row.GetField(j)->CompareEquals(row_kv.second->at(j)));
51     }
52     // free spaces
53     delete row_kv.second;
54 }
55 ASSERT_EQ(size, 0);

```

```

52 // 迭代器应当到达堆表尾部
53 ASSERT_TRUE(itr == table_heap->End());
54 }

```

Listing 3.8: TableIteratorTest

下图展示了框架给定的测试和自行设计的测试的通过情况：

图 3.3: 模块 2 测试通过情况

3.3 模块 3

B+ 树原理在《高级数据结构与算法》课程中有详细讲解，这里也不作展开赘述（天下 B+ 树都一个样），主要是涉及的部分比较多（之前两个模块都有涉及），调试比较复杂。

框架已经完整提供的是：IndexRootsPage、BPlusTreeIndex 这两个类。应当首先阅读这两个类，从 IndexRootsPage 中了解 BPlusTreePage 作为一个物理页的底层布局，从 BPlusTreeIndex 中了解 BPlusTree 的接口是如何调用的。除此以外，还需要阅读 GenericKey 和 KeyManager，它们的接口很简单，但需要学习怎么在 B+ 树 Key 相关操作中使用。了解了这些信息以后，我们就可以从构造函数开始设计 B+ 树了：

3.3.1 设计与实现

框架给出的 BPlusTree 内部接口众多，其实我不是很喜欢这样的设计，我认为 API 应当尽可能简洁、功能单一，比如我会选择把内部和外部页全部抽象成符号表等。但是因为框架各个部分关联紧密（涉及后续的并发控制等），不易进行接口上的更改，我也就没有自行设计。我绘制了一张图以梳理 BPlusTree 内部接口之间的调用关系，不同颜色表示不同类型。设计时我们自顶向下设计，但在实现时，应当从调用

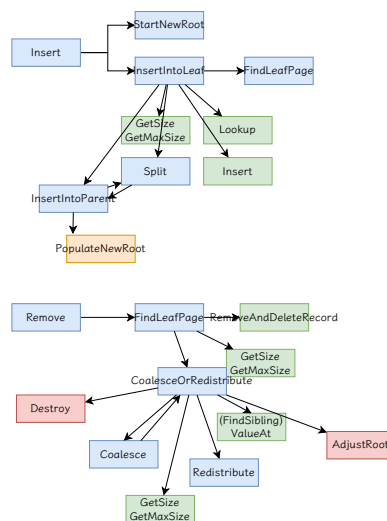


图 3.4: B+ 树内部接口调用关系

末端的函数开始实现，再逐次向上实现。否则，上层的函数写好之后发现下层的函数实现困难，就需要向上进行修改，这是非常麻烦的。

我们先来看 BPlusTree 的总体设计：

- 构造函数：
 - B+ 树元数据存储在各自的 `root_page`，但实际上在此之前还有 `IndexRootPage` 也算 B+ 树的元数据，毕竟每个 B+ 树就是一个 Index。
 - 因此构造函数首先操作 `IndexRootPage` 与 Index 建立关联，随后再将 `root_page` 设置为空，等待有数据插入时再真正创建树。
- `FindLeafPage`：该函数被多个地方调用，应当首先实现。实现简单，递归向下查询到叶子节点即可。
- `InsertIntoLeaf`、`Split`、`InsertIntoParent`：这几个函数参与插入操作，实现简单，直接调用 `BPlusTreePage` 中的对应操作，再负责处理页间连接的逻辑即可。以 `InsertIntoParent` 为例：
 - 首先，函数检查输入的旧节点是否是根节点。如果是，那么就需要创建一个新的根节点，并初始化它。然后，将旧节点和新节点的信息填充到新的根节点中，并更新根节点的 ID。最后，将旧节点和新节点的父节点 ID 设置为新的根节点的 ID，并将新的根节点的页从缓冲池中解锁。
 - 如果旧节点不是根节点，那么就需要找到旧节点的父节点，并在父节点中插入新节点的信息。然后，更新父节点中旧节点的键。如果父节点的大小小于其最大大小，那么就将父节点的页从缓冲池中解锁。
 - 如果父节点的大小大于或等于其最大大小，那么就需要对父节点进行分裂操作，并将分裂后的新父节点的信息插入到祖父节点中。这是一个递归的过程，直到找到一个不需要分裂的祖先节点为止。最后，将父节点和新父节点的页从缓冲池中解锁。
- `CoalesceOrRedistribute`：这是删除的核心操作，负责调整节点，也是我认为实现最困难的一个函数（从上面的调用图也可以看出来）。它可能遇到非常多种情况，需要调用 5 种不同的函数来处理。下面是我的设计：
 - 首先，函数获取输入节点的父节点，并找到输入节点在父节点中的索引位置。然后，根据索引位置，确定输入节点的邻居节点。
 - 接下来，函数检查输入节点和邻居节点的大小之和是否大于或等于输入节点的最大大小。如果是，那么就需要进行重新分配操作。在重新分配操作中，会将一部分键从一个节点移动到另一个节点，以保持两个节点的大小平衡。重新分配后，需要更新父节点中的键，以保持 B+ 树的有序性质。然后，函数会将邻居节点和父节点的页从缓冲池中解锁，并返回 `false`，表示没有发生删除操作。
 - 如果输入节点和邻居节点的大小之和小于输入节点的最大大小，那么就需要进行合并操作。在合并操作中，会将一个节点的所有键移动到另一个节点，并删除空的节点。如果父节点需要调整（例如，父节点的大小小于最小大小），那么就需要递归地对父节点进行合并或重新分配操作。
 - 如果父节点是根节点，并且需要删除，那么就需要调整根节点。最后，函数返回 `true`，表示发生了删除操作。
- `Coalesce`、`Remove`、`AdjustRoot`：这几个函数参与删除操作，同样是调用 `BPlusTreePage` 中的操作，并负责页间逻辑。这几个函数需要特别注意页间关系的保留与删除，很容易出现漏判的情况。以 `AdjustRoot` 为例：

- 首先，函数检查旧的根节点是否是叶子节点。如果是，并且旧的根节点的大小为 0（也就是说，根节点中没有任何元素），那么就需要更新根节点的 ID 为无效 ID，并调用 UpdateRootPageId 函数来更新根节点的 ID。然后，函数返回 true，表示根节点应该被删除。
- 如果旧的根节点不是叶子节点，那么就需要删除旧的根节点，并将其唯一的子节点设置为新的根节点。首先，函数获取旧的根节点的 ID 和新的根节点的 ID。然后，将新的根节点的 ID 设置为根节点的 ID，并获取新的根节点的页。接着，将新的根节点的父节点 ID 设置为无效 ID，并调用 UpdateRootPageId 函数来更新根节点的 ID。最后，函数返回 true，表示根节点应该被删除。

下面的代码清单给出了 FindLeafPage 函数的递归实现：

```

1 Page *BPlusTree::FindLeafPage(const GenericKey *key, page_id_t page_id, bool leftMost) {
2     auto page = reinterpret_cast<BPlusTreePage *>(buffer_pool_manager_>FetchPage(page_id)
3         ->GetData());
4     if (page->IsLeafPage()) {
5         return reinterpret_cast<Page *>(page);
6     }
7     auto internal_page = reinterpret_cast<InternalPage *>(page);
8     auto next_page_id = leftMost ? internal_page->ValueAt(0) : internal_page->Lookup(key,
9         processor_);
10    buffer_pool_manager_>UnpinPage(internal_page->GetPageId(), false);
11    return FindLeafPage(key, next_page_id, leftMost);
12 }

```

Listing 3.9: FindLeafPage 递归实现

接下来是 BPlusTreePage 的实现。这些操作与前两个模块相像，都涉及底层数据的移动等，可以利用框架提供的宏进行方便的操作，在此不展开赘述。下面的代码清单展示了内部页的 LoopUp 操作，使用二分查找进行加速：

```

1 page_id_t InternalPage::Lookup(const GenericKey *key, const KeyManager &KM) {
2     return INVALID_PAGE_ID;
3     int l = 1, r = GetSize();
4     while (l < r) {
5         int mid = (l + r) / 2;
6         if (KM.CompareKeys(KeyAt(mid), key) <= 0) {
7             l = mid + 1;
8         } else {
9             r = mid;
10        }
11    }
12    return ValueAt(l - 1);
13 }

```

Listing 3.10: 内部页二分查找

最后是索引迭代器的实现，这与前一模块的堆表迭代器相似，也不展开分析。下面的代码清单展示了迭代器的递增函数实现：

```

1 IndexIterator &IndexIterator::operator++() {
2     if (item_index + 1 < page->GetSize()) {
3         item_index++;
4     } else {
5         page_id_t next_page_id = page->GetNextPageId();

```

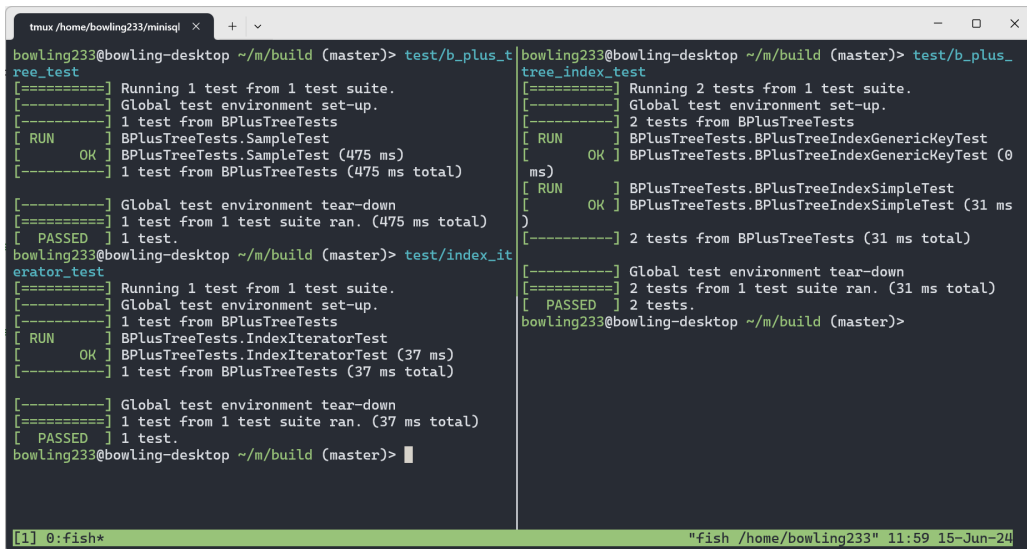


图 3.5: 模块 3 测试情况

```
6     buffer_pool_manager->UnpinPage(current_page_id, false);
7     current_page_id = next_page_id;
8     item_index = 0;
9     if (current_page_id != INVALID_PAGE_ID) {
10         page = reinterpret_cast<LeafPage *>(buffer_pool_manager->FetchPage(current_page_id)
11         ->GetData());
12     }
13     return *this;
14 }
```

Listing 3.11: 索引迭代器

3.3.2 测试

该模块有 3 个单元测试：

- `b_plus_tree_test`：将 `keys`、`values` 和 `delete_seq` 进行了随机打乱，然后将打乱后的键和值插入到 `tree` 中。检查插入、删除、查询是否正确。
- `b_plus_tree_index_test`：验证了 B+ 树索引的基本功能和通用键的操作，包括插入、扫描和迭代器功能。
- `index_iterator_test`：检查了 B+ 树索引迭代器在进行插入、删除、查找和遍历时的正确性和一致性

因为 B+ 树和索引对外封装非常简洁，操作就只有插入、删除、查找三个。上面的三个单元测试已经对索引和 B+ 树进行了完善的测试，核对了各步骤操作后数据的一致性。且该模块代码工作量已经比较大，因此我没有再为该模块设计更多的测试。

下图展示了框架提供的三个测试的通过情况：

第四章 模块 7

模块 7 的编程比较具有挑战性，虽然表面上去了解一下 `std::mutex` 就能会用锁，但多线程编程需要一定的经验和更多的知识才能确保无误。比如：锁的最佳实践不是调用成员函数 `lock()` 和 `unlock`，而是使用标准库提供的类模板 `std::lock_guard<>` 和模板参数推导。编程过程中我阅读了《C++ 并发编程实战（第 2 版）》中的相关章节，重点关注基于锁的并发数据结构，从而对框架的并发控制有了完善的理解。

4.1 LockManager

让我们仔细阅读已提供的数据结构中的成员，对它们使用做一些约定和说明：

- LockRequest：只有 `granted_` 了才表明锁真正被授予，这需要 LM 中的函数直接维护。
- LockRequestQueue
 - 提供锁请求的放置和清理服务。每个事务在队列中只能允许一个请求存在。这一点应当由事务自身保证。也就是说，对于一项数据，事务在获得锁后最多只能执行对共享锁的升级操作，而不应当在释放锁之前再提交锁申请。
 - 一旦 LM 发起锁的申请，立即将请求放入队列。只有 LM 的 `Unlock` 会从队列中删去请求，事务的 `Abort` 也是通过该方式删除请求的。事务一旦进入 `Abort`，那么队列中就不会存在该事务的请求（由 `TxnManager` 负责）。这样的接口设计使得，不需要在队列中查找请求是否存在，只需要查看事务状态就能获知。
 - 因为 `Abort` 可能会异步执行，因此每个申请锁的调用在等待条件满足后需要再次确认事务状态不是 `Abort`。
 - `is_upgrading`：一旦有合理的升级请求，就应当置位，以免后续任何锁继续进入造成饥饿。负责执行升级的函数负责将其复位，即使遇到异常。
 - `is_writing`：持有后才置位。删除写锁时复位。
 - `sharing_cnt_`：由 LM 负责维护。

几个数据结构之间的组合关系如下图所示：

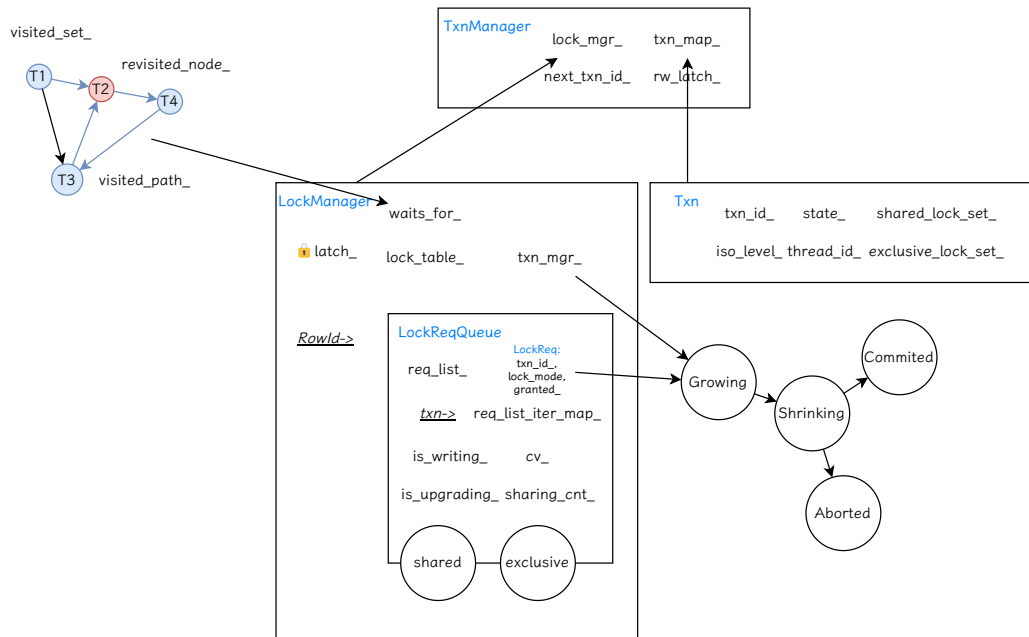


图 4.1: 并发控制相关的数据结构之间的关系

根据现有数据结构设计 LockManager 各个函数的行为：

- **LockShared**：共享锁请求。如果当前有排他锁，或者有升级请求，就应当等待。否则，直接授予共享锁。
- **LockExclusive**：排他锁请求。如果当前有排他锁、共享锁，或者有升级请求，就应当等待。否则，直接授予排他锁。
- **Unlock**：释放锁。无需等待直接释放，并调整队列状态。
- **LockUpgrade**：升级请求。如果有其他升级请求，抛出冲突升级异常。否则立即变更队列状态，等待当前排他锁、共享锁释放，然后授予排他锁。

上面的每个函数都需要检测事务状态、获得 LM 的互斥锁、检测队列状态并使用条件变量进行等待等，在此不做赘述。下面的代码展示了 LockUpdate 的实现：

```

1 bool LockManager::LockUpgrade(Txn *txn, const RowId &rid) {
2     // 异常：事务状态错误
3     if (txn->GetState() != TxnState::kGrowing) {
4         txn->SetState(TxnState::kAborted);
5         throw TxnAbortException(txn->GetTxnId(), AbortReason::kLockOnShrinking);
6     }
7     auto &lock_request_queue = lock_table_[rid];
8     // 异常：冲突升级
9     if (lock_request_queue.is_upgrading_) {
10        txn->SetState(TxnState::kAborted);
11        throw TxnAbortException(txn->GetTxnId(), AbortReason::kUpgradeConflict);
12    }
13    // 获取原有的锁
14    auto old_lock = lock_request_queue.GetLockRequestIter(txn->GetTxnId());
15    if (old_lock == lock_request_queue.req_list_.end()) {
16        throw "No lock to upgrade";
17    }

```

```

18 // 告知锁升级
19 std::unique_lock<std::mutex> lock(latch_);
20 lock_request_queue.is_upgrading_ = true;
21 old_lock->lock_mode_ = LockMode::kExclusive;
22 // 阻塞情况:
23 // 1. 有排他锁
24 // 因为有 UpgradeConflict, 所以必然不会有其他锁等待
25 // 3. 有共享锁
26 // 到 Upgrade 时, 应当保证共享锁是自己的
27 lock_request_queue.cv_.wait(lock, [&] {
28     // //DLOG(INFO) << "notified";
29     if (lock_request_queue.is_writing_ || lock_request_queue.sharing_cnt_ > 1) {
30         return false;
31     }
32     return true;
33 });
34 // 异常: 事务状态错误
35 if (txn->GetState() != TxnState::kGrowing) {
36     lock_request_queue.is_upgrading_ = false;
37     lock_request_queue.cv_.notify_all();
38     return false;
39 }
40 // 更新锁的类型
41 old_lock->granted_ = LockMode::kExclusive;
42 // 更新请求队列状态
43 lock_request_queue.sharing_cnt_--;
44 lock_request_queue.is_writing_ = true;
45 lock_request_queue.is_upgrading_ = false;
46 // 更新事务锁计数
47 txn->GetSharedLockSet().erase(rid);
48 txn->GetExclusiveLockSet().insert(rid);
49 // 通知等待
50 lock_request_queue.cv_.notify_all();
51 return true;
52 }

```

Listing 4.1: LockUpgrade

4.2 测试

让我们来分析已有的测试:

- **SLockInReadUncommittedTest**: 隔离级别为 RU, 在一条记录上创建一个共享锁。应当拒绝创建。共享锁用于防止其他事务在读取操作进行时修改数据, 在 RU 下无意义。
- **TwoPhaseLockingTest**: 测试单个事务。隔离级别为默认 (RR)。
 - 分别在两条记录上创建一个共享锁和一个排他锁。
 - 执行解锁操作, 事务状态发生变化, 说明由 LM 的锁函数控制事务状态。
 - 再次锁定, 抛出异常, shrink 阶段不允许锁定。
 - 事务 Abort, 释放锁。

- UpgradeLockInShrinkingPhase: 上一个测试的 Upgrade 版本。
- UpgradeConflictTest: 双线程测试。
 - 都创建共享锁
 - 线程 0 尝试 Upgrade, 须等待线程 1 释放。
 - 线程 1 尝试 Upgrade, 但线程 0 已经在请求, 冲突。
 - 线程 1 Abort, 锁释放。线程 0 继续执行。
- UpgradeTest: 简单的 Upgrade 测试。
- UpgradeAfterAbortTest: 两个事务均创建共享锁。其中一个事务尝试 Upgrade, 此时发生死锁 (等待另一个事务等待自己释放锁), Upgrade 返回, 并 Abort。
- BasicCycleTest: 死锁图结构测试。
- DeadlockDetectionTest: 死锁测试
 - 两个线程交互获取对方已独占的锁。后一个获取者造成死锁, 被 Abort, 前者继续执行并 Commit。
 - N 个事务和数据。首先全部创建共享锁。事务 2 创建 1 的共享锁。接下来启动并发: 事务 2 和 3 试图在 0 上创建排他锁, 需等待 0 释放。事务 1 请求 3 的排他锁, 需等待 3 释放。事务 0 请求 1 的排他锁, 需要等待 2 和 1 释放。并发请求完成后进入等待, 检测到死锁, 事务 2 和 3 被回滚。事务 0 得到 1 的排他锁, 并 commit 释放所有锁。事务 1 得到 3 的排他锁, 并 commit 释放所有锁。

我们可以补充如下测试:

- BulkUpdateTest: 大量事务并发请求升级锁。N 个事务在数据上创建共享锁, 然后依次 (并发) 尝试升级, 升级成功后 Commit 释放锁。应当只有第一个事务能够在所有后续事务尝试升级但因为冲突升级未能成功释放锁后成功升级并 commit。注意, 因为线程执行顺序不确定, 因此这个成功的线程不一定是线程 0。
- BulkTwoPhaseLockingTest: 大量二阶段锁事务并发请求。N 个事务在数据上随机创建共享锁或排他锁, 创建成功后等待一定时间 Commit 释放锁。应当所有事务都能够成功 Commit。值得注意的是, 应当观察到共享锁的请求全部立即成功, 因为即使共享锁的请求排在排他锁之后, 也不会发生冲突, 被优先授予。在上面的设计部分, 我们也说明了只应对了升级可能产生的饥饿。这个测试用例可以验证这一点。

下面的代码清单展示了 BulkUpdateTest 的实现:

```

1 TEST_F(LockManagerTest, BulkUpdateTest) {
2     // 准备死锁检测
3     auto cycle_detection_interval = std::chrono::milliseconds(500);
4     lock_mgr->EnableCycleDetection(cycle_detection_interval);
5     std::thread detect_worker(std::bind(&LockManager::RunCycleDetection, lock_mgr_));
6
7     RowId row(0, 0);
8     const uint32_t n = 1000;
9     std::vector<Txn *> txn(n);
10    for (uint32_t i = 0; i < n; i++) {

```

```

11     txn[i] = txn_mgr->Begin();
12     lock_mgr->LockShared(txn[i], row);
13 }
14 // 新建线程尝试升级
15 std::thread threads[n];
16 for (uint32_t i = 0; i < n; i++) {
17     threads[i] = std::thread([this, i, &row, &txn] {
18         DLOG(INFO) << "Thread " << i << " try to upgrade";
19         bool res;
20         try {
21             res = lock_mgr->LockUpgrade(txn[i], row);
22         } catch (TxnAbortException &e) {
23             ASSERT_EQ(AbortReason::kUpgradeConflict, e.abort_reason_);
24             ASSERT_EQ(TxnState::kAborted, txn[i]->GetState());
25             txn_mgr->Abort(txn[i]);
26             DLOG(INFO) << "Thread " << i << " aborted";
27             return;
28         }
29         ASSERT_TRUE(res);
30         txn_mgr->Commit(txn[i]);
31         ASSERT_EQ(TxnState::kCommitted, txn[i]->GetState());
32         DLOG(INFO) << "Thread " << i << " committed";
33     });
34 }
35 std::this_thread::sleep_for(cycle_detection_interval * 2);
36
37 // 等待线程结束
38 for (auto &t : threads) {
39     t.join();
40 }
41
42 lock_mgr->DisableCycleDetection();
43 detect_worker.join();
44 for (auto &t : txn) {
45     delete t;
46 }
47 }

```

Listing 4.2: BulkUpdateTest

设计中很多不完善的地方是在设计补充测试时才发现的，这也说明了测试的重要性。比如在设计 BulkTwoPhaseLockingTest 时，发生了上面共享锁全部立即成功的情况。这是因为我对 is_writing 的设计所致。从提供的测试中我发现队列中只能有一个 Upgrade，所以 is_upgrading 可以当成一个“互斥锁”来表征整个队列的状态。但是队列中可以有多个排他锁请求共存，is_writing 已经用于表示当前是否有授予的排他锁。如果要阻塞排他锁请求后的其他共享锁，则需要遍历队列或者再加一个标志位。因为更改基础变量的设计需要修改的代码较多，并且，这一不完善之处不会影响隔离级别为 RR 和 RC 的事务，而可能会影响到 RU，考虑到时间限制，我没有进行这个修改。

下图展示了框架给定的测试和自行设计的测试的通过情况：

```

bowling233@bowling-desktop ~/m/build (master)> test/7_test
[=====] Running 2 tests from 1 test suite.
[=====] Global test environment set-up.
[-----] 2 tests from LockManagerTest
[ RUN      ] LockManagerTest.BulkUpdateTest (1500 ms)
[ OK       ] LockManagerTest.BulkUpdateTest (1500 ms)
[ RUN      ] LockManagerTest.BulkTwoPhaseLockTest (25538 ms)
[ OK       ] LockManagerTest.BulkTwoPhaseLockTest (25538 ms)
[-----] 2 tests from LockManagerTest (27039 ms total)

[=====] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (27039 ms total)
[ PASSED ] 2 tests.
bowling233@bowling-desktop ~/m/build (master)>

bowling233@bowling-desktop ~/m/build (master)> test/lock_manager_test
[=====] Running 10 tests from 1 test suite.
[=====] Global test environment set-up.
[-----] 10 tests from LockManagerTest
[ RUN      ] LockManagerTest.SLockInReadUncommittedTest (0 ms)
[ OK       ] LockManagerTest.SLockInReadUncommittedTest (0 ms)
[ RUN      ] LockManagerTest.TwoPhaseLockingTest (0 ms)
[ OK       ] LockManagerTest.TwoPhaseLockingTest (0 ms)
[ RUN      ] LockManagerTest.UpgradeLockInShrinkingPhase (0 ms)
[ OK       ] LockManagerTest.UpgradeLockInShrinkingPhase (0 ms)
[ RUN      ] LockManagerTest.UpgradeConflictTest (100 ms)
[ OK       ] LockManagerTest.UpgradeConflictTest (100 ms)
[ RUN      ] LockManagerTest.UpgradeTest (0 ms)
[ OK       ] LockManagerTest.UpgradeTest (0 ms)
[ RUN      ] LockManagerTest.UpgradeAfterAbortTest (100 ms)
[ OK       ] LockManagerTest.UpgradeAfterAbortTest (100 ms)
[ RUN      ] LockManagerTest.BasicCycleTest1 (0 ms)
[ OK       ] LockManagerTest.BasicCycleTest1 (0 ms)
[ RUN      ] LockManagerTest.BasicCycleTest2 (0 ms)
[ OK       ] LockManagerTest.BasicCycleTest2 (0 ms)
[ RUN      ] LockManagerTest.DeadlockDetectionTest1 (1500 ms)
[ OK       ] LockManagerTest.DeadlockDetectionTest1 (1500 ms)
[ RUN      ] LockManagerTest.DeadlockDetectionTest2 (1101 ms)
[ OK       ] LockManagerTest.DeadlockDetectionTest2 (1101 ms)
[-----] 10 tests from LockManagerTest (2804 ms total)

[=====] Global test environment tear-down
[=====] 10 tests from 1 test suite ran. (2804 ms total)
[ PASSED ] 10 tests.
bowling233@bowling-desktop ~/m/build (master)>

```

图 4.2: LockManager 测试通过情况

4.3 思考题

其实，不把 LockManager 独立出来，其实就是把索引等数据结构实现成基于锁的并发数据结构，其中锁的粒度根据隔离级别而定。具体考虑到模块三的 B+ 树，我将做如下设计。

4.3.1 锁的粒度与隔离级别

在阅读代码时我们已经注意到，B+ 树的节点是在 BufferPoolManager 中管理的，因此 B+ 树本身其实没有放置线程锁的位置，直接利用 BufferPoolManager 和内存中 Page 的 `rw latch_`、`pin_count`、`latch_` 等成员即可。

隔离级别的实现与锁的持续时间和范围有关。下图来自 Alibaba Cloud Community，展示了不同隔离级别下的锁的粒度：

	Read Committed	Repeatable Read	Serializable
Select from table	Short duration Item S locks	Long duration Item S locks	Long duration Predicate/Table S locks
select from table for update; delete/update table ;	Long duration Item X locks	Long duration Item X locks	Long duration Predicate/Table X locks
delete/update from in(select from table); Insert into (select from table);	Short duration Item S locks	Long duration Item S locks	Long duration Predicate/Table S locks
Insert into table ;	Inserted item: Long duration, X locks Inserted Predicate: X Locks check		

图 4.3: 不同隔离级别下的锁的粒度

对应到 B+ 树的节点上，我们可以设计如下的锁的粒度：

- Table Lock：直接锁住 B+ 树对应的 BufferPoolManager。可以看到对于查询和更新，只有 serialize 级别会锁，实验中没有该级别。但是不论是什么级别，对 B+ 树的插入和删除因为会影响到整个树的结构，因此都需要 Table Lock。
- Item Lock：锁住对应记录的 Leaf Page。除了上面提到的情况都是 Item Lock。

锁的持续时间由事务管理模块负责，此处不赘述。

4.3.2 具体的 B+ 树操作设计

本节我们以 RR 隔离级别为例设计 B+ 树操作中的锁行为：

- 查询：在 BufferPoolManager 没有排他锁的情况下，给对应的 Leaf Page 上共享锁，直到事务结束。
- 插入、删除：为 BufferPoolManager 上排他锁，直到事务结束。
- 更新：在 BufferPoolManager 没有排他锁的情况下，对应的 Leaf Page 上排他锁，直到事务结束。

相关 B+ 树操作应当做的修改如下：

- 查询：修改 GetValue 函数，调用后立即等待 BufferPoolManager 的排他锁解锁，后等待 LeafPage 排他锁结束，添加共享锁，再进行 LookUp。
- 插入：修改 InsertIntoLeaf 函数，首先获取 BufferPoolManager 的排他锁，然后在 FindLeafPage 之后等待 LeafPage 上锁结束，无需获取 LeafPage 的排他锁（因为已持有更大范围的排他锁），插入数据。
- 删除：修改 Remove 函数，同插入。
- 更新：修改 Update 函数（虽然当前框架没有该函数，只需要用 FindLeafPage 简单包装即可），同查询，但添加的是排他锁。

Page 只有一个 rwlatch_，我们可以结合 pin_count 来标识共享锁和排他锁：0 为排他锁，大于 0 为共享锁，值为共享锁数量。

因为对 Page 实施了并发操作，BufferPoolManager 的相关操作也需要修改：

- UnpinPage：需要等待锁释放。

FlushPage、DeletePage 等操作都在 UnpinPage 后执行，因此无需再次确认锁的释放。

此外，Txn 需要修改，记录自己持有锁的信息（包括 Page 的锁和 BufferPoolManager 的锁），以便在事务结束时释放锁。只要保持接口不变，就不用修改 TxnManager，按原样负责在事务终结时释放事务持有的锁。

第三部分

个人详细报告

第五章 模块 4-6

5.1 模块 4

5.1.1 模块概述

该模块管理所有表和索引的信息，是实现 minisql 在命令行中运行的很重要的承上启下的一部分。一方面，向底层的缓冲池分配、删除或刷新数据页，联系列、行和 schema 得到表和索引；另一方面，它给 executor 的执行提供了接口，executor 通过调用该模块中的函数达到真正的数据库操作。

- 目录元信息

1. TableMeta

表的元信息中储存了该表的 id，名字，列的信息以及我们存储该表的元信息的数据页 id。该元信息可以在需要表信息时被反序列化，然后生成对应的 TableInfo（包含表的元信息和对应的 TableHeap）

2. IndexMeta

索引的元信息中储存了该索引的 id，名字，对应的元组以及对应的表 id。该元信息可以在需要索引信息时被反序列化，然后生成对应的 IndexInfo（包含了这个索引定义时的元信息 metadata，该索引对应的表信息 tableinfo，该索引的模式信息 keyschema 和索引操作对象 index）

3. CatalogMeta

数据对象 CatalogMeta 记录和管理表和索引的元信息被存储在哪个数据页中。CatalogMeta 的信息将会被序列化到数据库文件的第 CATALOG META PAGE ID 号数据页中（逻辑意义上），CATALOG META PAGE ID 默认值为 0。

- 表和索引的管理

- CatalogManager 类应具备维护和持久化数据库中所有表和索引的信息。CatalogManager 能够在数据库实例（DBStorageEngine）初次创建时（init = true）初始化元数据；并在后续重新打开数据库实例时，从数据库文件中加载所有的表和索引信息，构建 TableInfo 和 IndexInfo 信息置于内存中。此外，CatalogManager 类还需要对上层模块提供对指定数据表的操作方式，如 CreateTable、GetTable、GetTables、DropTable、GetTableIndexes；对上层模块提供对指定索引的操作方式，如 CreateIndex、GetIndex、DropIndex。
- Catalog Manager 负责管理和维护数据库的所有模式信息，包括：
 - * 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
 - * 表中每个字段的定义信息，包括字段类型、是否唯一等。
 - * 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

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

5.1.2 具体实现

- 序列化

参考之前 RecordManager 中的序列化和反序列化，需要通过魔数 MAGIC NUM 来确保序列化和反序列化的正确性。本节中只要实现上述三种元信息的 GetSerializedSize()。实现过程比较简单这里不赘述。可以通过观察 SerializeTo()，即返回其中每个对象的长度，这里注意 vector 和 string 的类型长度是固定的，即 sizeof(string) 为 4 个 B，需要特别注意。

- 构造函数

- 该函数在数据库第一次创建时 (init 是 1)，初始化元数据。
- 该函数在重新打开数据库时，从数据库文件中加载所有的表和索引信息，构建 TableInfo 和 IndexInfo 信息置于内存中。
 - * 该操作首先要从缓冲池中读取 CatalogMeta 对应的数据页的信息并反序列化为 CatalogMeta
 - * 然后加载所有表的信息，该操作需要从 CatalogMeta 得到所有表的元信息的页码，然后读取所有表的元信息，对每一个元信息，将 table name 和 table 添加入原有的 map，为了得到 table，我们需要得到 table heap 来初始化对应的 Table Info。
 - * 最后加载所有索引的信息，与上述过程一致，读取索引元信息，然后更新 index name 和 index。
- 注意：meta page 是脏页。

- Create Table

- 若表已经存在，则不能创建同名的表
- 更新 table id 和 table name
- 创建新表的元信息数据页和数据结构，并更新目录元信息（脏页）
- 更新 table

- Create Index

- 若表不存在，则不能在表上建立索引
- 若索引已经存在，则不能创建同名索引
- 若创建索引对应的列不存在，则不能对其索引
- 更新 index id 和 index name
- 创建新索引的元信息数据页和数据结构，并更新目录元信息（脏页）
- 更新 index

- Get Table

- 得到对应名字的表，只需检查是否存在这样的表，若存在则返回 success
- 得到所有表，只需遍历 tables，将 Table Info 的 vector 返回即可

- Get Index

- 得到对应表对应的索引，只需检查是否存在这样的表，若存在则检查这个表上是否存在这样的索引，都存在则返回 Index Info 即可

- 得到所有对应表上的索引，只需检查是否存在这样的表，若存在则返回该表对应的所有索引 Index Info
- **Drop Table**
 - 若该表不存在则不能 drop
 - 在 table name 和 tables 中删除该表，并在 Catalog Meta 中删除 table 对应的元信息页
 - 在缓冲池中删除 table 对应的元信息页
- **Drop Index**
 - 若该表不存在或表上的索引不存在则不能 drop
 - 在 index name 和 indexes 中删除该表，并在 Catalog Meta 中删除 index 对应的元信息页
 - 在缓冲池中删除 index 对应的元信息页
- **Load Table**
 - 若该表已经存在则不能添加同名表
 - 从 CatalogMeta 得到表的元信息的页码，然后读取表的元信息，将 table name 和 table 添加入原有的 map，为了得到 table，我们需要得到 table heap 来初始化对应的 Table Info。
 - 其实 Load Table 应该为构造函数的一个部分
- **Load Index**
 - 若该索引已经存在则不能添加同样索引
 - 与上述过程一致，读取索引元信息，然后更新 index name 和 index。
 - 其实 Load Index 也应该为构造函数的一个部分
- **FlushCatalogMetaPage**

该函数实际作用为在析构时将现在的目录元信息序列化为字符串保存到磁盘中，以便下一次使用时重载本来的数据。需要利用 Buffer Pool 中的 Flush Page 重写信息。

5.1.3 测试

- 已有测试
 - CatalogMetaTest
非常简单的测试，仅仅测试了其保存表和索引元信息的功能
 - CatalogTableTest
该测试新建一个表，检测新建表的正确性，然后测试了 Get Table 的功能，另外还关闭数据库后重新打开，再 Get Table 可以看到正确加载表
 - CatalogIndexTest
该测试新建一个表，在此基础上新建一个索引，测试新建索引的正确性，在建好索引的表上插入多条新数据，并搜索这些数据观察正确性。另外也关闭数据库后重新打开，通过搜索上述数据的正确性来检测加载 Index 是否成功
- 添加测试观察上述测试，实际上基本完备，只是对每个函数的检测的情况较少。以下测试新增了对 Get Tables, Get Index, Get Table Index, Drop Index, Drop Table 的测试，并且对上述的 Create Table, Create Index 增加新的情况的测试。但对于 Index 的插入和搜索的测试我认为原有测试已经能够说明，故不多补充。


```

1 TEST(CatalogTest, CatalogAllTest) {
2     //test for init catalog
3     auto db_01 = new DBStorageEngine(db_file_name, true);
4     auto &catalog_01 = db_01->catalog_mgr_;
5
6     //create table test
7     TableInfo *table_info = nullptr;
8     std::vector<Column *> columns = {new Column("id", TypeId::kTypeInt, 0, false,
9         false),
10                                     new Column("name", TypeId::kTypeChar, 64, 1, true
11         , false),
12                                     new Column("account", TypeId::kTypeFloat, 2, true
13         , false)}};
14     auto schema = std::make_shared<Schema>(columns);
15     Txn txn;
16     catalog_01->CreateTable("table-1", schema.get(), &txn, table_info);
17     ASSERT_TRUE(table_info != nullptr);
18     ASSERT_EQ(DB_TABLE_ALREADY_EXIST, catalog_01->CreateTable("table-1", schema.get(),
19         &txn, table_info));
20
21     //get table test
22     TableInfo *table_info_02 = nullptr;
23     ASSERT_EQ(DB_SUCCESS, catalog_01->GetTable("table-1", table_info_02));
24     ASSERT_EQ(table_info, table_info_02);
25     TableInfo *table_info_03 = nullptr;
26     ASSERT_EQ(DB_TABLE_NOT_EXIST, catalog_01->GetTable("table-2", table_info_03));
27     TableInfo *table_info_04 = nullptr;
28     std::vector<Column *> columns1 = {new Column("id", TypeId::kTypeInt, 0, false,
29         false),
30                                     new Column("name", TypeId::kTypeChar, 50, 1, true
31         , false)}};
32     schema = std::make_shared<Schema>(columns1);
33     catalog_01->CreateTable("table-2", schema.get(), &txn, table_info_04);
34     vector<TableInfo*> tableinfo_vector;
35     catalog_01->GetTables(tableinfo_vector);
36     vector<TableInfo*> tableinfo_temp;
37     tableinfo_temp.push_back(table_info_04);
38     tableinfo_temp.push_back(table_info);
39     ASSERT_EQ(tableinfo_vector, tableinfo_temp);
40
41     //create index test
42     IndexInfo *index_info = nullptr;
43     std::vector<std::string> bad_index_keys{"id", "age", "name"};
44     std::vector<std::string> index_keys{"id", "name"};
45     std::vector<std::string> index_keys_pri{"id"};
46     auto r1 = catalog_01->CreateIndex("table-0", "index-0", index_keys, &txn,
47         index_info, "bptree");
48     ASSERT_EQ(DB_TABLE_NOT_EXIST, r1);
49     auto r2 = catalog_01->CreateIndex("table-1", "index-1", bad_index_keys, &txn,
50         index_info, "bptree");
51     ASSERT_EQ(DB_COLUMN_NAME_NOT_EXIST, r2);
52     auto r3 = catalog_01->CreateIndex("table-1", "index-1", index_keys, &txn,

```

```

        index_info, "bptree");
45  ASSERT_EQ(DB_SUCCESS, r3);
46  auto r4 = catalog_01->CreateIndex("table-1", "index-1", index_keys, &txn,
        index_info, "bptree");
47  ASSERT_EQ(DB_INDEX_ALREADY_EXIST, r4);
48
49  //get index test
50  IndexInfo *index_info_02;
51  catalog_01->GetIndex("table-1", "index-1", index_info_02);
52  ASSERT_EQ(index_info, index_info_02);
53  IndexInfo *index_info_03;
54  auto r5 = catalog_01->CreateIndex("table-1", "primary", index_keys_pri, &txn,
        index_info_03, "bptree");
55  ASSERT_EQ(DB_SUCCESS, r5);
56  vector<IndexInfo*> indexinfo_vector;
57  vector<IndexInfo*> indexinfo_temp;
58  indexinfo_temp.push_back(index_info_03);
59  indexinfo_temp.push_back(index_info);
60  catalog_01->GetTableIndexes("table-1", indexinfo_vector);
61  ASSERT_EQ(indexinfo_vector, indexinfo_temp);
62
63  delete db_01;
64
65  //test loading
66  auto db_02 = new DBStorageEngine(db_file_name, false);
67  auto &catalog_02 = db_02->catalog_mgr_;
68  TableInfo *table_info_13 = nullptr;
69  ASSERT_EQ(DB_TABLE_NOT_EXIST, catalog_02->GetTable("table-3", table_info_13));
70  ASSERT_EQ(DB_SUCCESS, catalog_02->GetTable("table-1", table_info_13));
71  IndexInfo *index_info_13;
72  ASSERT_EQ(DB_SUCCESS, catalog_02->GetIndex("table-1", "primary", index_info_13));
73
74  //drop index test
75  ASSERT_EQ(DB_INDEX_NOT_FOUND, catalog_02->DropIndex("table-1", "index-2"));
76  ASSERT_EQ(DB_TABLE_NOT_EXIST, catalog_02->DropIndex("table-3", "index-1"));
77  ASSERT_EQ(DB_SUCCESS, catalog_02->DropIndex("table-1", "primary"));
78
79  //drop table test
80  ASSERT_EQ(DB_TABLE_NOT_EXIST, catalog_02->DropTable("table-3"));
81  ASSERT_EQ(DB_SUCCESS, catalog_02->DropTable("table-1"));
82  TableInfo *table_info_23 = nullptr;
83  ASSERT_EQ(DB_TABLE_NOT_EXIST, catalog_02->GetTable("table-1", table_info_23));
84  delete db_02;
85 }

```

Listing 5.1: 4 test

测试结果如下:

```

gsy@Gsy:~/minisql/build$ ./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 (47 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[          OK ] CatalogTest.CatalogIndexTest (37 ms)
[-----] 3 tests from CatalogTest (84 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test suite ran. (85 ms total)
[ PASSED ] 3 tests.

```

图 5.1: catalog test

```

gsy@Gsy:~/minisql/build$ ./test/4_test
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from CatalogTest
[ RUN      ] CatalogTest.CatalogAllTest
[          OK ] CatalogTest.CatalogAllTest (89 ms)
[-----] 1 test from CatalogTest (90 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (90 ms total)
[ PASSED ] 1 test.

```

图 5.2: 4 test

5.2 模块 5

5.2.1 模块概述

该模块主要包括 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 (Identifier) 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。而这个部分框架均已完成, 我们只需知道每种操作的语法树即可。Executor 遍历查询计划树, 将树上的 PlanNode 替换成对应的 Executor, 即得到具体的操作, 随后调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行, 并执行结果返回给上层模块。这部分实际上就是我们在命令行中输入语句后, 负责执行语句的总控制模块。

• Planner

1. 语法树数据结构

- 语法树每个结点都包含了一个唯一标识符 id, 唯一标识符在调用 CreateSyntaxNode 函数时生成。type 表示语法树结点的类型, line no 和 col no 表示该语法树结点对应的是 SQL 语句的第几行第几列, child 和 next 分别表示该结点的子结点和兄弟结点, val 用作一些额外信息的存储 (如在 kNodeString 类型的结点中, val 将用于存储该字符串的字面量)。
- 每个节点都有一个 Type, 这个类型对应所有支持实现的语句, 也即对应 executor 中不同的执行。类型有如下几种, 具体使用将在之后介绍: kNodeUnknown, kNodeQuit, kNodeExecFile, kNodeIdentifier, kNodeNumber, kNodeString, kNodeNull, kNodeCreateDB, kNodeDropDB, kNodeShowDB, kNodeUseDB, kNodeShowTables, kNodeCreateTable, kNodeDropTable, kNodeShowIndexes, kNodeInsert, kNodeDelete, kNodeUpdate, kNodeSelect, kNodeConditions, kNodeConnector, kNodeCompareOperator, kNodeColumnType,

kNodeColumnDefinition, kNodeColumnDefinitionList, kNodeColumnList, kNodeColumnValues, kNodeUpdateValues, kNodeUpdateValue, kNodeAllColumns, kNodeCreateIndex, kNodeDropIndex, kNodeIndexType, kNodeTrxBegin, kNodeTrxCommit, kNodeTrxRollback

2. 生成查询计划

- 对于复杂的语句，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 column 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 Planndoe 交由 executor 进行执行。
- 为了理解语法树，可以运行 ./bin/main，此时会在 build 下看到一系列名为 syntax tree i.txt 的文件，通过观察这些文件的内容，可以得出语法树的结构，我绘制这些图，便于下一个部分的使用。

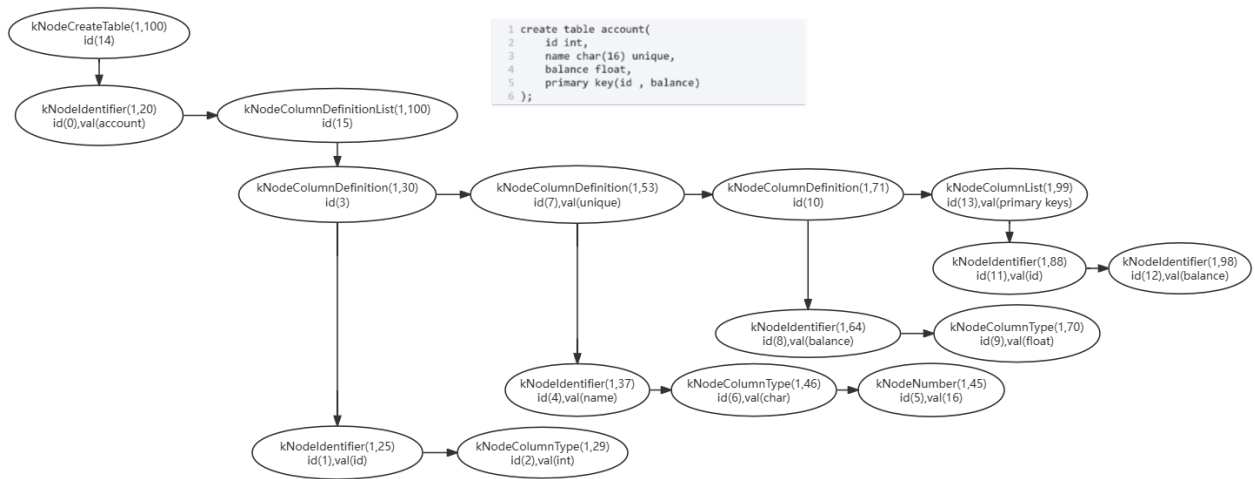


图 5.3: create table

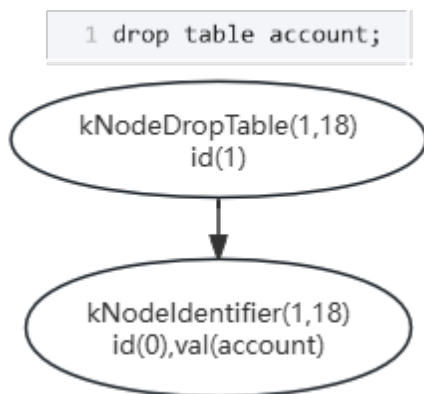


图 5.4: drop table



图 5.5: show indexes

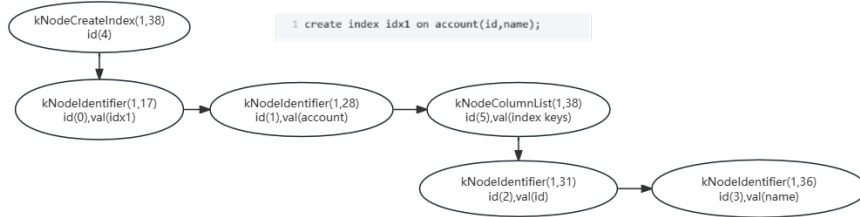


图 5.6: create index

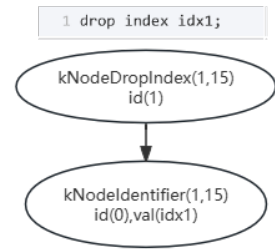


图 5.7: drop index

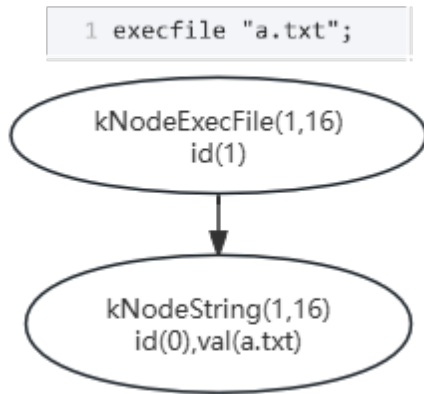


图 5.8: execfile

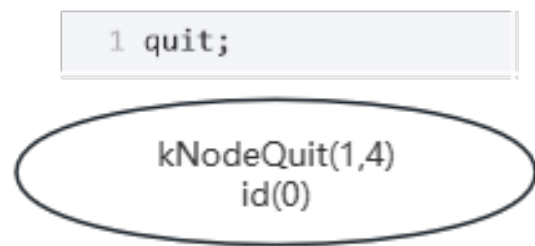


图 5.9: quit

• Executor

1. 在拿到 Planner 生成的具体的查询计划后，就可以生成真正执行查询计划的一系列算子了。生成算子的步骤很简单，遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor。
2. 算子的执行模型为 Iterator Model，即经典的火山模型。执行引擎会将整个 SQL 构建成一个 Operator 树，查询树自顶向下的调用接口，数据则自底向上的被拉取处理。每一种操作会抽象为一个 Operator，每个算子都有 Init() 和 Next() 两个方法。Init() 对算子进行初始化工作。Next() 则是向下层算子请求下一条数据。当 Next() 返回 false 时，则代表下层算子已经没有剩余数据，迭代结束。
3. 在本次任务中，我们将实现 5 个算子，分别是 select, Index Select, insert, update, delete。对于每个算子，都实现了 Init 和 Next 方法。Init 方法初始化运算符的内部状态，Next 方法提供迭代器接口，并在每次调用时返回一个元组和相应的 RID。对于每个算子，我们假设它在单线程上下文中运行，并不需要考虑多线程的情况。每个算子都可以通过访问 ExecuteContext 来实现表的修改，例如插入、更新和删除。为了使表索引与底层表保持一致，插入删除时还需要更新索引。

5.2.2 具体实现

• ExecuteCreateTable

- 这个函数是该模块里最复杂的一个函数，我们在新建表时，还要考虑 primary key 和 unique 约束，即在建表的同时会建对应的索引，另外还考虑了 primary key 必须满足 unique 条件，所以

primary key 也会建立一个 unique 索引，请关注上一部分中的图理解本部分内容（之后不再赘述）

- 得到表的名字，和表的每一列，利用这两个信息可以新建表
- 需要注意的是，primary key 和 unique 实际上不在同一种结点下，所以我提前处理了 primary key 的信息，即保存所有 primary key 的列名
- 遍历所有的列，只有 Unique 约束和 primary key 的列保存在 Unique 中，并在新建表之后，新建 unique 和 primary key 索引。

- **ExecuteDropTable**

这一部分比较简单，只要调用 catalog manager 中的 drop table 即可，但需注意，当表被丢弃时，其上的索引都应被丢弃，所以在 drop table 成功后，应该把所有在其上的索引都删除，即调用 catalog manager 中的 drop index。

- **ExecuteShowIndexes**

- 该函数重在理解函数的意思，show 即在终端展示所有的 Index, 包含当前数据库中所有的 Table 中的所有 Index。这里需要我们得到所有的 table，并用其 table name 得到所有的 index。
- 在这里我采用了按照 table 的顺序依次输出所有的索引的方式。

- **ExecuteCreateIndex**

该函数也很简单，我们需要从语法树中得到 table name, index name, 然后再得到所有的索引对应的列，然后采用 catalog manager 中的 create index 创建索引。

- **ExecuteDropIndex**

这个函数也重在理解，我们只会得到 index 的名字，所以我们需要从 catalog manager 中得到所有的 table，然后我们找到 index 对应的 table，并将该索引删除。

- **ExecuteExecfile**

其实对于该函数，我并不是很了解其中具体的执行细节，只是在 main 函数中，通过读取终端输入的内容，生成对应的语法树，并交由 executor 执行；该函数实际与 main 一样，只是读取文件的内容，生成对应的语法树，并交由 executor 执行。

- **ExecuteQuit**

该函数只需删除数据库并返回 DB QUIT 即可。

5.2.3 测试

由于该模块实际都在 main 中体现，所以我选择采用不同的测试方法，对所有给定被实现的语句，将其整合在一个文件中，利用 Execfile，逐步实现上述操作，通过观察结果就能得出模块是否正确。

```
1 create database db1;
2 show databases;
3 drop database db1;
4 show databases;
5 create database db0;
6 use db0;
7 show tables;
8 create table t1(a int, b char(20) unique, c float, primary key(a, c));
9 create table student(sno char(8), sage int, sab float unique, primary key (sno, sab))
;
```

```

10  show tables;
11  drop table student;
12  show tables;
13  create index idx1 on t1(a, b);
14  show indexes;
15  drop index idx1;
16  show indexes;
17  insert into t1 values(1, "aaa", 2.33);
18  insert into t1 values(2, "bbb", 42.3);
19  insert into t1 values(3, "ccc", 60.5);
20  select * from t1;
21  update t1 set b = "mmm";
22  select * from t1;
23  update t1 set a = 1, b = "ccc" where c = 2.33;
24  select * from t1;
25  select a, b from t1;
26  select * from t1 where a = 1;
27  select * from t1 where a > 1 and a < 3;
28  delete from t1 where a = 1 and c = 2.33;
29  select * from t1;
30

```

Listing 5.2: mytest.txt

原有的测试实际上与本模块关系并不大，只做了很少的测试，且测试部分均已实现，所以以下仅展示测试结果测试结果如下：

```

minisql > execfile "mytest.txt";
[INFO] Sql syntax parse ok!
+-----+
| Database |
+-----+
| db1      |
+-----+
Empty set (0.00 sec)
Database changed
+-----+
| Tables_in_db0 |
+-----+
+-----+
+-----+
| Tables_in_db0 |
+-----+
| student      |
| t1           |
+-----+

```

图 5.10: 5 test 1

```

+-----+
| Tables_in_db0 |
+-----+
| t1            |
+-----+
Show Indexes
      table: t1
            index: AUTO_CREATED_INDEX_OF_a_c_ON_t1
            index: UNIQUE_c_ON_t1
            index: idx1
            index: UNIQUE_b_ON_t1
            index: UNIQUE_a_ON_t1
5 index(es) have listed.
Index not exists.
Show Indexes
      table: t1
0 index(es) have listed.
Query OK, 1 row affected(0.0000 sec).
Query OK, 1 row affected(0.0000 sec).
Query OK, 1 row affected(0.0000 sec).
+-----+
| a | b | c |
+-----+
| 1 | aaa | 2.330000 |
| 2 | bbb | 42.299999 |
| 3 | ccc | 60.500000 |
+-----+
3 row in set(0.0000 sec).
Query OK, 3 row affected(0.0000 sec).

```

图 5.11: 5 test 2

```

+---+-----+-----+
| a | b | c |
+---+-----+-----+
| 1 | mmm | 2.330000 |
| 2 | mmm | 42.299999 |
| 3 | mmm | 60.500000 |
+---+-----+-----+
3 row in set(0.0000 sec).
Query OK, 1 row affected(0.0000 sec).
+---+-----+-----+
| a | b | c |
+---+-----+-----+
| 1 | ccc | 2.330000 |
| 2 | mmm | 42.299999 |
| 3 | mmm | 60.500000 |
+---+-----+-----+
3 row in set(0.0000 sec).
+---+-----+
| a | b |
+---+-----+
| 1 | ccc |
| 2 | mmm |
| 3 | mmm |
+---+-----+
3 row in set(0.0000 sec).

```

图 5.12: 5 test 3

```

3 row in set(0.0000 sec).
+---+-----+-----+
| a | b | c |
+---+-----+-----+
| 1 | ccc | 2.330000 |
+---+-----+-----+
1 row in set(0.0000 sec).
+---+-----+-----+
| a | b | c |
+---+-----+-----+
| 2 | mmm | 42.299999 |
+---+-----+-----+
1 row in set(0.0000 sec).
Query OK, 1 row affected(0.0000 sec).
+---+-----+-----+
| a | b | c |
+---+-----+-----+
| 2 | mmm | 42.299999 |
| 3 | mmm | 60.500000 |
+---+-----+-----+
2 row in set(0.0000 sec).

```

图 5.13: 5 test 4

```

gsy@Gsy:~/minisql/build$ ./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 (68 ms)
[ RUN ] ExecutorTest.SimpleDeleteTest
[ OK ] ExecutorTest.SimpleDeleteTest (51 ms)
[ RUN ] ExecutorTest.SimpleRawInsertTest
[ OK ] ExecutorTest.SimpleRawInsertTest (41 ms)
[ RUN ] ExecutorTest.SimpleUpdateTest
[ OK ] ExecutorTest.SimpleUpdateTest (50 ms)
[-----] 4 tests from ExecutorTest (211 ms total)

[-----] Global test environment tear-down
[=====] 4 tests from 1 test suite ran. (212 ms total)
[ PASSED ] 4 tests.

```

图 5.14: executor test

5.3 模块 6

5.3.1 模块概述

- Recovery Manager 负责管理和维护数据恢复的过程，包括：日志结构的定义；检查点 CheckPoint 的定义；执行 Redo、Undo 等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态
- 第一阶段，数据库启动后，InnoDB 会通过 redo log 找到最近一次 checkpoint 的位置，然后根据 checkpoint 相对应的 LSN 开始，获取需要重做的日志，接着解析日志并且保存到一个哈希表中，最后通过遍历哈希表中的 redo log 信息，读取相关页进行恢复。

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

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

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

5.3.2 具体实现

- **LogRec**

- **GetPrevLSN**

- prev lsn map 存着从事务到前一个线程的 map，当事务在 map 中时，其前一个线程为对应的值，并用当前线程替换前一个线程，当事务不在 map 中，就将当前线程存入，在该事务的下一个线程使用函数时，会返回当前的线程。

- **CreateInsertLog**

- 返回一个 Insert 的日志，该线程就在当前线程上 +1，并用其前一个线程初始化它（即始终记录该事务的前一条）。该日志对应的 key 和 val 分别为插入内容的 key 和 val。

- **CreateDeleteLog**

- 返回一个 Delete 的日志，该线程就在当前线程上 +1，并用其前一个线程初始化它（即始终记录该事务的前一条）。该日志对应的 key 和 val 分别为删除内容的 key 和 val。

- **CreateUpdateLog**

- 返回一个 Create 的日志，该线程就在当前线程上 +1，并用其前一个线程初始化它（即始终记录该事务的前一条）。该日志保存旧的 Key 和 val 以及新的 key 和 val。

– **CreateBeginLog**

返回一个 Begin 的日志，该线程就在当前线程上 +1，并用其前一个线程初始化它（即始终记录该事务的前一条）。该日志不需要保存值。

– **CreateCommitLog**

返回一个 Commit 的日志，该线程就在当前线程上 +1，并用其前一个线程初始化它（即始终记录该事务的前一条）。该日志不需要保存值。

– **CreateAbortLog**

返回一个 Abort 的日志，该线程就在当前线程上 +1，并用其前一个线程初始化它（即始终记录该事务的前一条）。该日志不需要保存值。

• **RecoveryManager**

– **Rollback**

从当前活跃事务中得到回滚的第一个线程，当前的日志是 insert delete 和 update 的时候需要回滚事务，如果是 insert 则把插入的内容删除，如果是 delete 则把删除的内容重新插入，如果是 update 则把新的内容换为旧的，然后回滚它的前一个线程，直到所有线程都回滚完结束。

– **RedoPhase**

从 CheckPoint 开始，把后面的所有线程重新做一遍，如果是 Inset 就插入，Delete 就删除，Update 就更新，Commit 需要把该事务从活跃事务中删除，Abort 需要回滚事务。

– **UndoPhase**

把所有的正在活跃的事务都回滚，并将其从活跃事务列表中删除。

5.3.3 测试

由于该模块其实并无实际意义，且实际只做了 Redo 和 Undo 两个部分的工作，观察给的测试，我们发现它几乎把所有情况的 Redo 和 Undo 情况都测过了，所以在此不补充额外测试，仅将原给测试分为两部分单独测试 Redo 和 Undo，其结果与本来的结果相比，能够看出 Redo 的实际存在意义。

```
1 //从checkpoint开始Redo后再Undo的结果
2 recovery_mgr.UndoPhase();
3 ASSERT_EQ(db["A"], 2000);
4 ASSERT_EQ(db["B"], 1000);
5 ASSERT_EQ(db["C"], 600);
6 ASSERT_EQ(db.count("D"), 0);
```

Listing 5.3: recovery manager test.txt

```
1 //未Redo直接Undo的结果
2 recovery_mgr.UndoPhase();
3 ASSERT_EQ(db["A"], 2000);
4 ASSERT_EQ(db["B"], 1000);
5 ASSERT_EQ(db.count("C"), 0);
6 ASSERT_EQ(db.count("D"), 0);
```

Listing 5.4: 6 test.txt

```

gsy@Gsy:~/minisql/build$ ./test/6_test
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from RecoveryManagerTest
[ RUN      ] RecoveryManagerTest.RedoTest
[ OK       ] RecoveryManagerTest.RedoTest (0 ms)
[ RUN      ] RecoveryManagerTest.UndoTest
[ OK       ] RecoveryManagerTest.UndoTest (0 ms)
[-----] 2 tests from RecoveryManagerTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (0 ms total)
[ PASSED  ] 2 tests.

```

图 5.15: 6 test

```

gsy@Gsy:~/minisql/build$ ./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.

```

图 5.16: recovery manager test

5.3.4 思考题

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？

在回答问题之前，首先复习及了解一下 ARIES 的机制

- LSN

name	where	definition
flushedLSN	memory	最后落盘的那个 LSN
pageLSN	buffer pool page	与该页相关的最新 LSN
recLSN	buffer pool page	在上次落盘之后，与该页相关的最老 LSN
lastLSN	transaction	某事务最后一条日志的 LSN
MasterRecord	disk	最近一次 checkpoint 的 LSN

- 事务操作

- 事务提交

当事务提交时，DBMS 先写入一条 COMMIT 记录到 WAL，然后将 COMMIT 及之前的日志落盘。一旦 COMMIT 记录安全地存储在磁盘上，DBMS 就向应用程序返回事务已提交的确认信息，并将 flushedLSN 被修改为 COMMIT 记录的 LSN。在将来某一时刻，DBMS 会将内存中 COMMIT 及其之前的日志清除，并将缓存池中的脏页刷回磁盘，完成后再写入一条 TXN-END 记录到 WAL 中，作为内部记录。需要注意的是 TXN-END 不需要立刻落盘。对于执行提交的事务来说，COMMIT 与 TXN-END 之间没有别的操作。

- 事务中止

处理事务回滚，必须从 WAL 中找出所有与该事务相关的日志及其执行顺序。由于在 DBMS 中执行的所有事务的操作记录都会写到 WAL 中，因此为了提高效率，同一个事务的每条日志中需要记录上一条记录的 LSN，即 prevLSN。一个特殊情况是：第一条 BEGIN 记录的 prevLSN 为空。

- CLR

CLR 记录的是 undo 操作，它除了记录原操作相关的记录，还记录了 undoNext 指针，指向下一个将要被 undo 的 LSN。CLR 本身也是操作记录，因此它也需要像其它操作一样写进 WAL 中。直到 DBMS 通知应用程序该事务已经被终止的时刻，DBMS 不需要等待 CLR 落盘，

- **Fuzzy checkpoints**

- CHECKPOINT-BEGIN: checkpoint 的起点
- CHECKPOINT-END: checkpoint 的终点，同时包含 ATT (活动事务表) 和 DPT (脏页表) 记录。

当 checkpoint 成功完成时，CHECKPOINT-BEGIN 记录的 LSN 才被写入到数据库的 MasterRecord 中，任何在 checkpoint 之后才启动的事务不会被记录在 CHECKPOINT-END 的 ATT 中。

- **恢复阶段**

- **Analysis Phase**

从最近的 BEGIN-CHECKPOINT 开始往近处扫描日志：

- * 如果发现 TXN-END 记录，则从 ATT 中移除该事务
- * 遇到其它日志记录时，将事务放入 ATT 中，将 status 设置为 UNDO，如果事务提交，将其状态修改为 COMMIT
- * 如果是数据更新记录，按需更新 DPT 以及 recLSN

当 Analysis Phase 结束时：

- * ATT 告诉 DBMS 在发生故障时，哪些事务是活跃的
- * DPT 告诉 DBMS 在发生故障时，哪些脏数据页可能尚未写入磁盘

- **Redo Phase**

从 DPT 中找到最小的 recLSN，从那里开始重做更新记录和 CLR，重做时需要执行：

- * 重新执行日志中的操作
- * 将 pageLSN 修改成日志记录的 LSN
- * 不再新增操作日志，也不强制刷盘

在 Redo Phase 结束时，会为所有状态为 COMMIT 的事务写入 TXN-END 日志，同时将它们从 ATT 中移除。

- **Undo Phase**

将所有 Analysis Phase 判定为需要 Undo 状态的事务的所有操作按执行顺序倒序撤销，并且为每个 undo 操作写一条 CLR。

我对该问题的理解如下：

- **Recovery Manager**

- **Log Manager** 本结构在该模块中实际是未实现的，只是定义了简单的日志结构，如果要想让日志真正作为数据库的日志，需要对前面的模块进行一些小修改，实现 ARIES 架构。

- * **Disk Manager:** 由于日志是需要被记录在磁盘里的，所以我们的磁盘不止要实现数据库对应页的分配和存储，读入和写入，还要实现日志对应页的分配和存储，日志的写入和读入。
- * **Buffer Pool Manager:** 与磁盘几乎一致，我们的缓冲池应该在替换页等操作上作简要修改，使其不止能对数据页处理，还能适应日志。

- * Page: 由于我们要记录与页相关的 LSN, 所以应当给 Page 增加新的关于 LSN 的函数。
- * Transaction: 由于框架并未实现事务, 所以要实现对应的事务, 事务开始, 事务中止和事务提交都应该引起对应的日志写入。
- * Get/Set Lsn: 在上述知识模块中对应的 lsn 都要设置其读入和写入的函数。

除去上述修改的部分, Log Manager 需要包含以下的部分以及实现以下的操作

- * Log Manager 中应有一个缓冲区, 可以将新添入的日志写入缓冲区, 并能够在合适的时机写入磁盘。
- * Flush: 应有一个函数用来将日志写入缓冲区, 与 catalog manager 中使用 buffer pool manager 的 flush page 的思想应差不多。
- * StartFlush: 日志内容从缓冲区到磁盘需要在一定条件下发生: 缓冲池的剩余空间不足以插入新的记录; 缓冲池换出了一个脏页。满足这两个条件就将内容换到磁盘。
- * EndFlush: 当数据库系统被关闭时, 我们应该停止日志线程, 同时将缓冲区中的记录全部保存到硬盘中。
- * AddRecord: 在对应的时机应该能够插入新的记录, 关于新记录的生成在本模块中已经实现。(在事务开始、提交和中止以及数据的插入、删除和更新时添加)

此外还需对当前的日志进行修改, 日志可以进行序列化和反序列化便于从磁盘写入和读入。

– Recovery Manager

- * Recovery Manager 中应该能够得到上一次的 checkpoint, 含有一个 ATT 用来记录活跃事务列表。
- * Analyse: 从最近的 checkpoint 开始往近处扫描日志, 如果事务已经提交则从 ATT 移除, 如果发现了其他日志记录, 先将其放入 ATT 并设置为需要 Undo, 如果日志内容为更新, 则要更新脏页表和 recLSN。
- * Redo: 从 checkpoint 开始重做日志对应的内容。
- * Undo: 遍历 ATT 中的每一个事务, 对事务的操作进行回滚

• CheckPoint

- 在一定的情况下触发 checkpoint 机制。
- 记录当前活动的事务: 系统记录所有当前活动的事务, 即那些已经开始但尚未提交或回滚的事务。包括事务对应的信息, 以便在系统恢复时知道哪些事务需要被重做或撤销。
- 将所有已修改但尚未持久化到磁盘的缓冲区数据强制写入磁盘。这确保了在检查点之前的所有数据更改都被保存, 使得在发生系统崩溃时, 可以从这个点安全地恢复数据。该部分在恢复中会进行, 所以可以采取在一定情况下取最老的 modified page(last checkpoint) 对应的 LSN。
- 写入检查点日志记录, 实际上就是传入这个 checkpoint 等到需要恢复时使用该检查点。
- 可以选择合适的触发 checkpoint 的机制: 如定时触发, 或在缓冲区内存满后触发, 尤其是在系统关闭时应该触发。

5.4 Bonus: Clock replacer

5.4.1 概述

Clock Replacer 算法的思想很简单, 就像钟表的指针一样一圈一圈遍历, 直到找到符合替换的位置。先做出如下假设, buffer 的大小是 3, 访问的序列是 1, 2, 3, 1, 4, 3。最初的三个元素 1, 2, 3 会直接加入到 buffer 中, 如下表所示

num	ref bit	clock hand
1	1	<-
2	1	
3	1	

当访问到第二个 1 时，因为 1 在 buffer 中，直接读取。下面访问 4，这时发现 buffer 中并没有 4，那么就需要找一个位置替换并存入 4。首先会将 1 处的 ref bit 置 0，并且指针下移一位，如下表。

num	ref bit	clock hand
1	0	
2	1	<-
3	1	

现在，clock hand 指向的 2，而 ref bit 仍然为 1，则这一位也不能替换，那么就把对应的 ref bit 置 0，指针继续下移，如下表。

num	ref bit	clock hand
1	0	
2	0	
3	1	<-

同理，指针指向 3 遇到的情况与上次相同，因此继续执行相同操作。但是 clock hand 到达边界，需返回到初始位置。便得到如下状态

num	ref bit	clock hand
1	0	<-
2	0	
3	0	

此时 clock hand 指向的 ref bit 为 0，该位置能够被替换，替换 1 加入 4，并把 ref bit 置 1，得到如下状态

num	ref bit	clock hand
4	1	<-
2	0	
3	0	

最后访问元素 3，发现 buffer 中缓存有 3，直接输出即可。

5.4.2 实现

- **构造函数**

给定最大页数，初始化 clock list 为最大页数长，且每个元素都是一个 invalid frame id，且指针指向第一个元素。

- **Victim**

从指针位置开始，如果当前位置 ref 为 1，则将其设为 0 并移到下一位，如果当前位置 ref=0，则该页可被换出，这里注意从当前 list 中把该页删除，如果当前位置被 pin，则不能被换出；如果当前位置什么都没有，即 Invalid frame id，就移到下一位，如果指针位置越界，则返回 list 的开头，模拟时钟循环。

- **Pin**

只要存在这个页，就把它状态设为 Pin

- **Unpin**

如果当前 list 里有这页，直接修改其状态即可，现在的 ref 应为 1，如果 List 没有这一页，则需要把这页插到 list 中，只要找到一个不合法的页把其插入即可。

5.4.3 测试

采用与 Lru replacer 一样的测试，测试 Victim, Unpin, Pin 的正确性，

```
1 TEST(ClockReplacerTest, SampleTest) {
2     CLOCKReplacer clock_replacer(7);
3
4     // Scenario: unpin six elements, i.e. add them to the replacer.
5     clock_replacer.Unpin(1);
6     clock_replacer.Unpin(2);
7     clock_replacer.Unpin(3);
8     clock_replacer.Unpin(4);
9     clock_replacer.Unpin(5);
10    clock_replacer.Unpin(6);
11    clock_replacer.Unpin(1);
12    EXPECT_EQ(6, clock_replacer.Size());
13
14    // Scenario: get three victims from the clock.
15    int value;
16    clock_replacer.Victim(&value);
17    EXPECT_EQ(1, value);
18    clock_replacer.Victim(&value);
19    EXPECT_EQ(2, value);
20    clock_replacer.Victim(&value);
21    EXPECT_EQ(3, value);
22
23    // Scenario: pin elements in the replacer.
24    // Note that 3 has already been victimized, so pinning 3 should have no effect.
25    clock_replacer.Pin(3);
26    clock_replacer.Pin(4);
27    EXPECT_EQ(2, clock_replacer.Size());
28
29    // Scenario: unpin 4. We expect that the reference bit of 4 will be set to 1.
30    clock_replacer.Unpin(4);
31
32    // Scenario: continue looking for victims. We expect these victims.
33    clock_replacer.Victim(&value);
34    EXPECT_EQ(5, value);
35    clock_replacer.Victim(&value);
36    EXPECT_EQ(6, value);
37    clock_replacer.Victim(&value);
38    EXPECT_EQ(4, value);
39 }
40
```

Listing 5.5: mytest.txt

测试结果如下：

替换出的 Victim 都是正确的，且 pin 和 Unpin 的工作均正确。

```
gsy@Gsy:~/minisql/build$ ./test/clock_replacer_test
[=====] 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.
```

图 5.17: clock replacer test