Database System
Laboratory Projects

MiniSQL Report

```
[==========] Running 27 tests from 9 test suites.
[----------] Global test environment set-up.
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (8020 ms)
[----------] 1 test from BufferPoolManagerTest (8020 ms total)
```

```
[----------] 1 test from LRUReplacerTest
[ RUN      ] LRUReplacerTest.SampleTest
[       OK ] LRUReplacerTest.SampleTest (42 ms)
[----------] 1 test from LRUReplacerTest (42 ms total)
```

```
[----------] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (359 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (339 ms)
[----------] 3 tests from CatalogTest (699 ms total)
```

```
$ ./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 (240 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (254 ms)
[----------] 2 tests from BPlusTreeTests (495 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (495 ms total)
[  PASSED  ] 2 tests.
```

```
[       OK ] BPlusTreeTests.LeafRedistributionToRight (251 ms)
[ RUN      ] BPlusTreeTests.LeafRedistributionToLeaf
Intnl4id=2|: 0,(2)  5,(3)

Leaf2k=4:1,(0,1)  2,(0,2)  3,(0,3)  4,(0,4)
Leaf3k=3:5,(0,5)  6,(0,6)  7,(0,7)

Intnl4id=2|: 0,(2)  4,(3)

Leaf2k=3:1,(0,1)  2,(0,2)  3,(0,3)
Leaf3k=2:4,(0,4)  5,(0,5)

[       OK ] BPlusTreeTests.LeafRedistributionToLeaf (255 ms)
[----------] 9 tests from BPlusTreeTests (9894 ms total)

[----------] Global test environment tear-down
[==========] 9 tests from 1 test suite ran. (9894 ms total)
[  PASSED  ] 9 tests.
```

```
$ ./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 (77 ms)
[ RUN      ] DiskManagerTest.FreePageAllocationTest
[       OK ] DiskManagerTest.FreePageAllocationTest (32789 ms)
[----------] 2 tests from DiskManagerTest (32867 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (32867 ms total)
[  PASSED  ] 2 tests.
```

```
$ ./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 (240 ms)
[----------] 1 test from BPlusTreeTests (240 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (240 ms total)
[  PASSED  ] 1 test.
```

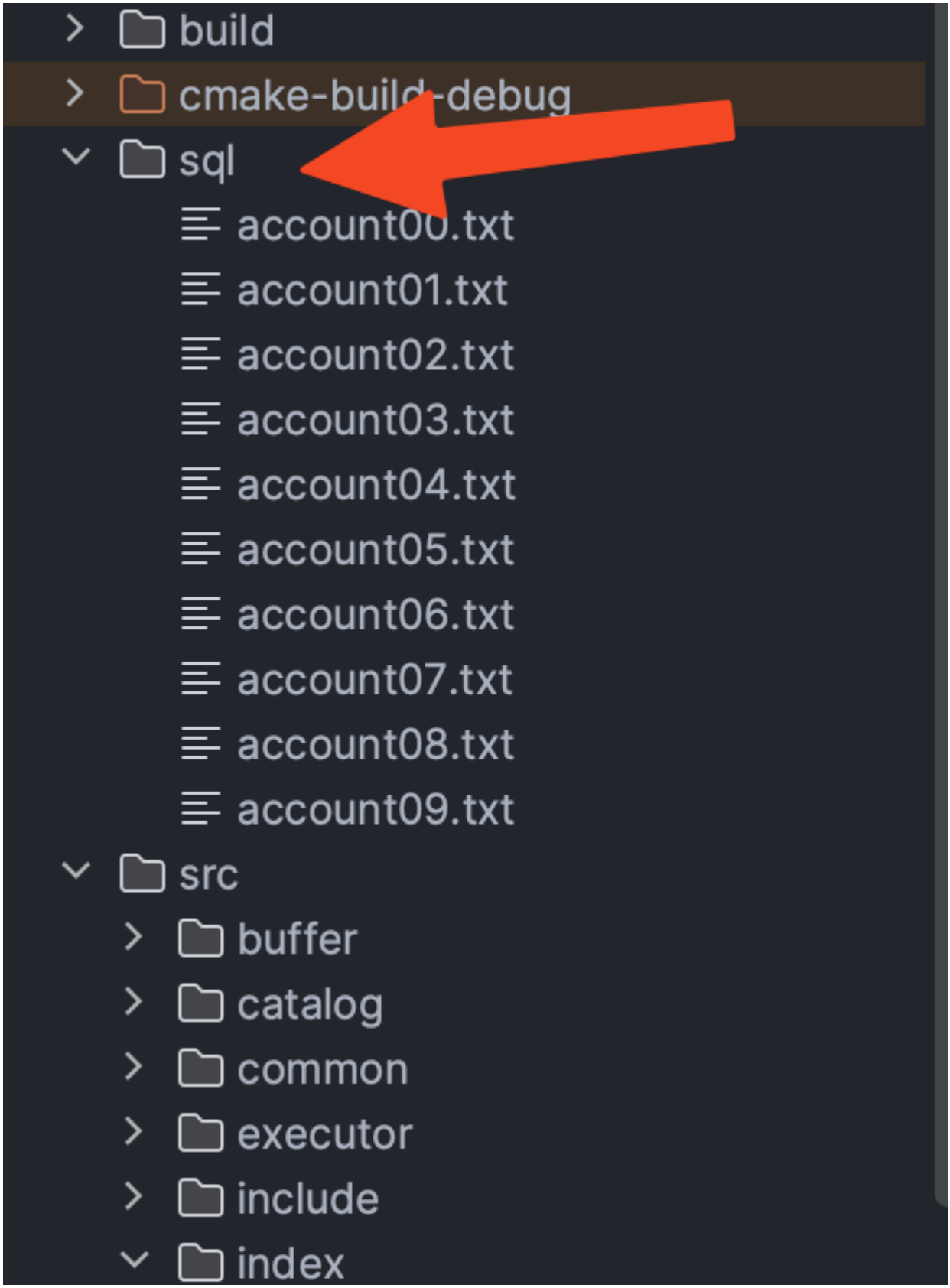
```
$ ./test/index_roots_page_test;
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from PageTests
[ RUN      ] PageTests.IndexRootsPageTest
[       OK ] PageTests.IndexRootsPageTest (0 ms)
[----------] 1 test from PageTests (0 ms total)

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

```
$ ./test/table_heap_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (7072 ms)
[----------] 1 test from TableHeapTest (7073 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (7073 ms total)
[  PASSED  ] 1 test.
```

```
$ ./test/tuple_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TupleTest
[ RUN      ] TupleTest.SerializeDeserializeTest
[       OK ] TupleTest.SerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.RowTest
[       OK ] TupleTest.RowTest (1 ms)
[----------] 2 tests from TupleTest (1 ms total)

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

线下验收（6.27晚）

使用的这里的sql语句，验收通过无问题（创表，select，index，drop等）



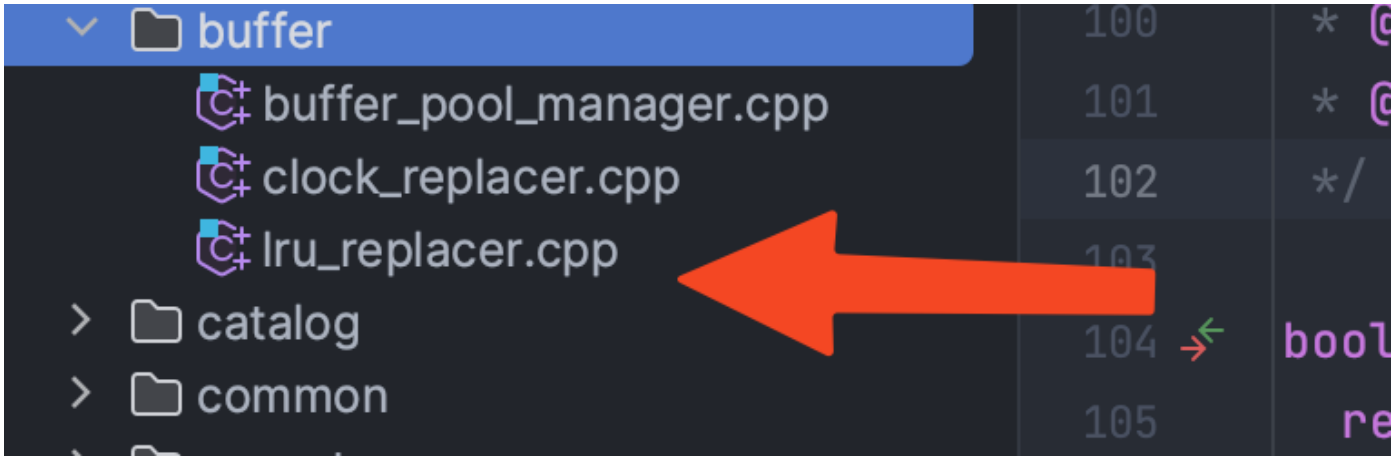
纸上写问题上交

Bonus

模块一的bonus：

Bonus：除LRU Replacer外，实现一种新的缓冲区替换算法（如Clock Replacer）。

实现在src和include对应文件下



具体内容在报告第一模块涉及

```
// Initialization
Create a circular linked list of pages
Set reference bit of all pages to 0
Set pointer (clock hand) to head of linked list

// Page Replacement
while (true) {
    if (current_page->reference_bit == 0) {
        // Select current page as replacement candidate
        selected_page = current_page
        current_page->reference_bit = 1
        break
    }
    current_page->reference_bit = 0
    current_page = current_page->next
}

// Replace Selected Page
ReplacePage(selected_page) // Replace selected page with new page to be brought into
buffer pool
UpdateMetadata(selected_page) // Update necessary data structures and metadata
```

分工

实验目的和需求

实验目的

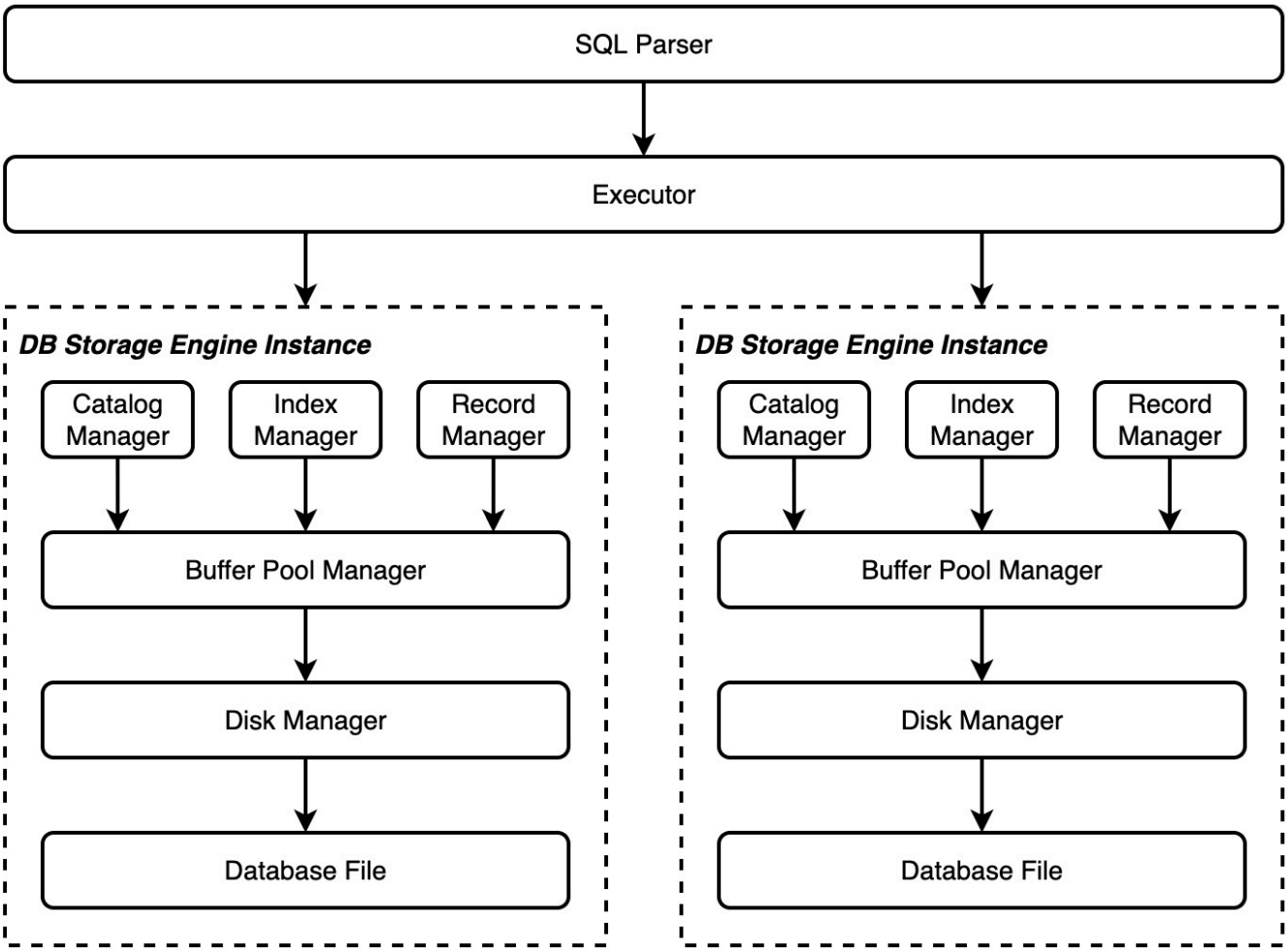
1. 设计并实现一个精简型单用户SQL引擎MiniSQL，允许用户通过字符界面输入SQL语句实现基本的增删改查操作，并能够通过索引来优化性能。
2. 通过对MiniSQL的设计与实现，提高学生的系统编程能力，加深对数据库管理系统底层设计的理解。

实验需求

1. 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
2. 表定义：一个表可以定义多达32个属性，各属性可以指定是否为`unique`，支持单属性的主键定义。
3. 索引定义：对于表的主属性自动建立B+树索引，对于声明为`unique`的属性也需要建立B+树索引。
4. 数据操作: 可以通过`and`或`or`连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。
5. 在工程实现上，使用源代码管理工具（如Git）进行代码管理，代码提交历史和每次提交的信息清晰明确；同时编写的代码应符合代码规范，具有良好的代码风格。

框架介绍

图



模块概述

Disk Manager

- Database File（DB File）是存储数据库中所有数据的文件，其主要由记录（Record）数据、索引（Index）数据和目录（Catalog）数据组成（即共享表空间的设计方式）。与书上提供的设计（每张表通过一个文件维护，每个索引也通过一个文件维护，即独占表空间的设计方式）有所不同。共享表空间的优势在于所有的数据在同一个文件中，方便管理，但其同样存在着缺点，所有的数据和索引存放到一个文件中将会导致产生一个非常大的文件，同时多个表及索引在表空间中混合存储会导致做了大量删除操作后可能会留有有大量的空隙。在本实验中，为了方便同学们实现，我们采取共享表空间的设计方式，即将所有的数据和索引放在同一个文件中。学有余力的同学可以额外尝试使用独占表空间的设计方式进行设计。
- Disk Manager负责DB File中数据页的分配和回收，以及数据页中数据的读取和写入。
- 对应实验：#1 DISK AND BUFFER POOL MANAGER。

Buffer Pool Manager

- Buffer Manager 负责缓冲区的管理，主要功能包括：
    1. 根据需要，从磁盘中读取指定的数据页到缓冲区中或将缓冲区中的数据页转储（Flush）到磁盘；
    2. 实现缓冲区的替换算法，当缓冲区满时选择合适的数据页进行替换；
    3. 记录缓冲区中各页的状态，如是否是脏页（Dirty Page）、是否被锁定（Pin）等；

4. 提供缓冲区页的锁定功能，被锁定的页将不允许替换。

- 为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是数据页（Page），数据页的大小应为文件系统与磁盘交互单位的整数倍。在本实验中，数据页的大小默认为 4KB。
- 对应实验：#1 DISK AND BUFFER POOL MANAGER。

Record Manager

- Record Manager 负责管理数据表中记录。所有的记录以堆表（Table Heap）的形式进行组织。Record Manager 的主要功能包括：记录的插入、删除与查找操作，并对外提供相应的接口。其中查找操作返回的是符合条件记录的起始迭代器，对迭代器的迭代访问操作由执行器（Executor）进行。
- 堆表是由多个数据页构成的链表，每个数据页中包含一条或多条记录，支持非定长记录的存储。不要求支持单条记录的跨页存储（即保证所有插入的记录都小于数据页的大小）。堆表中所有的记录都是无序存储的。
- 需要额外说明的是，堆表只是记录组织的其中一种方式，除此之外，记录还可以通过顺序文件（按照主键大小顺序存储所有的记录）、B+树文件（所有的记录都存储在B+树的叶结点中，MySQL中InnoDB存储引擎存储记录的方式）等形式进行组织。学有余力的同学可以尝试着使用除堆表以外的形式来组织数据。
- 对应实验：#2 RECORD MANAGER。

Index Manager

- Index Manager 负责数据表索引的实现和管理，包括：索引（B+树等形式）的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。
- B+树索引中的节点大小应与缓冲区的数据页大小相同，B+树的叉数由节点大小与索引键大小计算得到。
- 对应实验：#3 INDEX MANAGER。

Catalog Manager

- Catalog Manager 负责管理数据库的所有模式信息，包括：
  1. 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。
  2. 表中每个字段的定义信息，包括字段类型、是否唯一等。
  3. 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。
- Catalog Manager 还必需提供访问及操作上述信息的接口，供执行器使用。
- 对应实验：#4 CATALOG MANAGER。

Planner and Executor

- Planner（执行计划生成器）的主要功能是根据解释器（Parser）生成的语法树，通过Catalog Manager 提供的信息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，随后将这些词语转换成可以理解的各种 c++ 类。解析完成后，Planner根据改写语法树后生成的Statement结构，生成对应的Plannode，并将Plannode交由Executor进行执行。
- Executor（执行器）的主要功能是遍历Planner生成的计划树，将树上的 PlanNode 替换成对应的 Executor，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行。Executor采用的是火山模型，提供迭代器接口，每次调用时会返回一个元组和相应的 RID，直到执行完成。
- 语法树的相关结构请参考#5 PLANNER AND EXECUTOR。

SQL Parser

- 程序流程控制，即“启动并初始化 → ‘接收命令、处理命令、显示命令结果’循环 → 退出”流程。
- 接收并解释用户输入的命令，生成命令的内部数据结构表示，同时检查命令的语法正确性和部分语义正确性，对正确的命令生成语法树，然后调用执行器层提供的函数执行并显示执行结果，对不正确的命令显示错误信息。

第一模块报告

Principles & Theories

In the design of MiniSQL, the Disk Manager and Buffer Pool Manager are the modules at the lower level of the architecture, whose functionality and management capabilities directly affect the performance of the database, storage management and data access efficiency.

The Disk Manager is responsible for the management of database files, including the allocation and deallocation of data pages, and the reading and writing of data. **The Buffer Pool Manager** is responsible for moving data pages from memory to disk and providing read and write operations to data pages. Disk Manager and Buffer Pool Manager are transparent to other modules and provide a uniform interface for other modules to use.

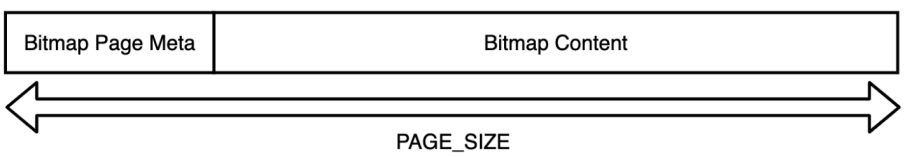
💡Summarizing the guiding reference, we face the implementions of the following modules:

- Bitmap Page
- Disk Manager
- Implementation of the buffer pool replacement strategy
- The LRU replacement
- The Clock replacement
- Buffer pool

Bitmap Page

This module acts as the warm-up, helping us better understand the managing strategy of disk in miniSQL.

Each bitmap page consists of two parts, one for meta information, which contains the allocated page 'page_allocated' and the next free page 'next_free_page_', the second part is a string of bits to indicate whether the page is allocated or not, 1 means allocated, 0 for not.



To implement this feature we define a class `BitmapPage`, with member functions `AllocatePage`, `DeAllocatePage`, `IsPageFree`.

In `bitmap_page.h` file we notice that

```
std::bitset<8 * MAX_CHARS> allocate_pages;
```

`std::bitset` is a fixed-size bitset container provided by the C++ standard library, with which `allocate_pages` functions as a bitmap to indicate how the data pages are allocated. Bitset provides some useful member functions to work with bit collections.

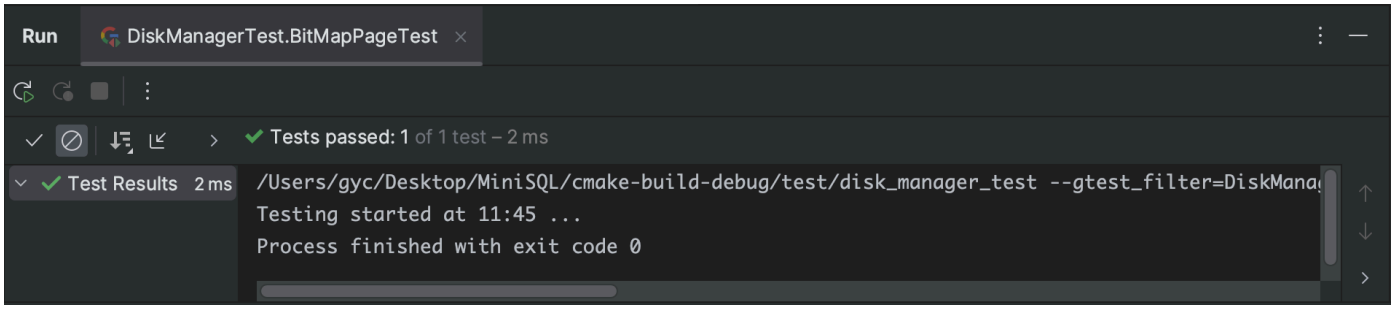
1. `std::bitset::size`：Returns the size of the bitset, that is, the total number of bits.
2. `std::bitset::count`：counts the number of bits in the bitset set to 1.
3. `std::bitset::test`：checks whether a bit at a particular position in the bitset is set to 1.
4. `std::bitset::set`：sets the bit at a specific position in the bitset to 1.

With these, implementing allocating function in the scale of bitmap becomes way more easier. Here we take `AllocatePage`'s implementation as example.

```
template<size_t PageSize>
bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset) {
  if (page_offset >= 8 * MAX_CHARS || allocate_pages.test(page_offset) ) { //if
page_offset is out of range or already allocated
    return false; //fail to allocate
  } else {
    allocate_pages.set(page_offset, 1); // set the position of page_offset to 1
    return true;
  }
}
```

Moving to `test/storage/disk_manager_test.cpp`, we run Google Test. The result shows that the test is passed.

💻Test Result：



> 📒NOTE:
>
> You might encounter a warning indicates that you should call the `testing::InitGoogleTest()` function to initialize before calling `RUN_ALL_TESTS()`. This is a enforced requirement of GTest in recent releases.
> So add code to initialize the Google Test framework,

```
GTEST_API_ int main(int argc, char **argv) {
testing::InitGoogleTest(&argc, argv);
return RUN_ALL_TESTS();
}
```

> Then you can successfully go through the test. ☑️ Note that this part should be deleted in the following tests.

Disk Manager

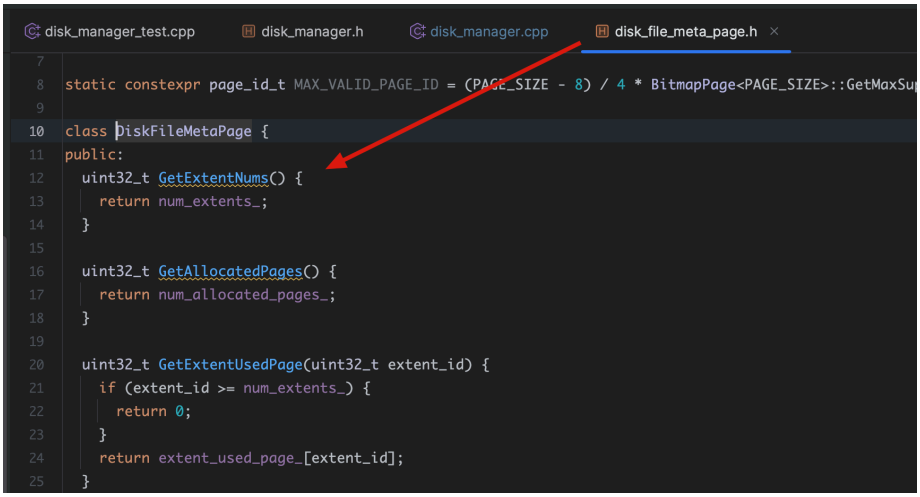
We add a continuous segment of data pages on the basis of a bitmap page which indicates allocaiton infos about those pages. To deal with the possible overflow problem, consider the above composite structure as an EXTENT of the database file, and then record the information of these partitions through an higher-level meta-information page.

Disk Meta Page	Extent Meta (Bitmap Page)	Extent Pages	Extent Meta (Bitmap Page)	Extent Pages	...

📌 Several notes to pay attention to

- expansion in scale of pages being maintained
- Discontinuity of the actual data pages
- Encapsulation of page allocation for Buffer Management : Mapping

Based on the `disk_file_meta_page.h`, we designed an algorithm to do the job of `AllocatePage`.



🧾 This algorithm abstracts the process into the following schema

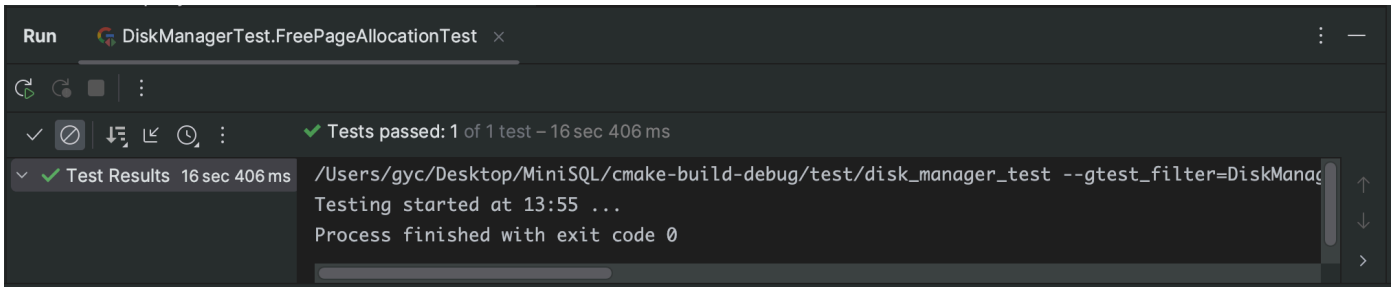
1. Retrieve the file's metadata page and the total number of extents.
2. Iterate through the extents, starting from the first one, until a non-full extent is found or all extents have been traversed.
3. If a non-full extent is found, update the bitmap (common case):
    - Write the updated bitmap page back to the disk.
    - Return the logical ID of the allocated page.

4. If all extents are full, try to create a new extent:
    - Check if there is available space to create a new extent.
    - If space is available
        - Write the new bitmap page to the disk.
        - Return the logical ID of the allocated page.
    - If there is no available space:
        - Return an invalid page ID.

The DeallocatePage method is basically the reserved version of the allocation.

- Find `logical_page_id`'s actual, read from disk
- Release the corresponding data page
- Write back to disk, update the metadata

💻 **Test Result：**



Buffer Pool Replacement

LRU

The Buffer Pool Replacer is responsible for keeping track of the usage of data pages in the Buffer Pool and deciding which data page to replace if there are no free pages in the Buffer Pool.

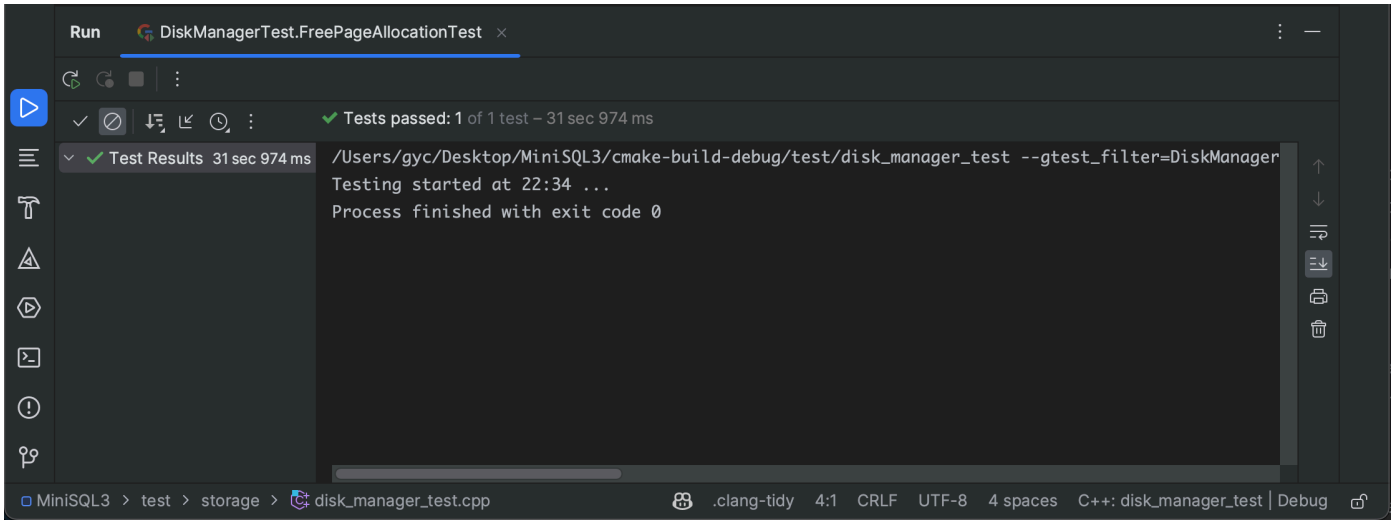
We designed `LRUReplacer` to keep track of the order in which data pages are accessed by using `visit_lst`.

🔧 Details about `visit_lst`

- `visit_lst` is a doubly linked list that maintains the pages to be replaced, which are in order of data pages' access. When a data page is accessed, it is moved to the **end** of the linked list, indicating that it is the most recently accessed page(so, head for LRU).
- When a data page needs to be replaced, the first node of the list can be removed from the head of the list, which represents the least recently accessed page.

By manipulating the access list and mapping table, to be specific, removing `victim` from head & evicting to `pin` & pushing back to `unpin`, LRUReplacer is able to do the job.

💻 Test Result：



Buffer Pool

The Buffer Pool Manager is responsible for fetching data pages from the Disk Manager and storing them into memory, and dumping dirty pages to disk if necessary (to make room for new pages). `Replacer` is used to decide which pages to replace when there are no free pages in the Buffer Pool for allocation.

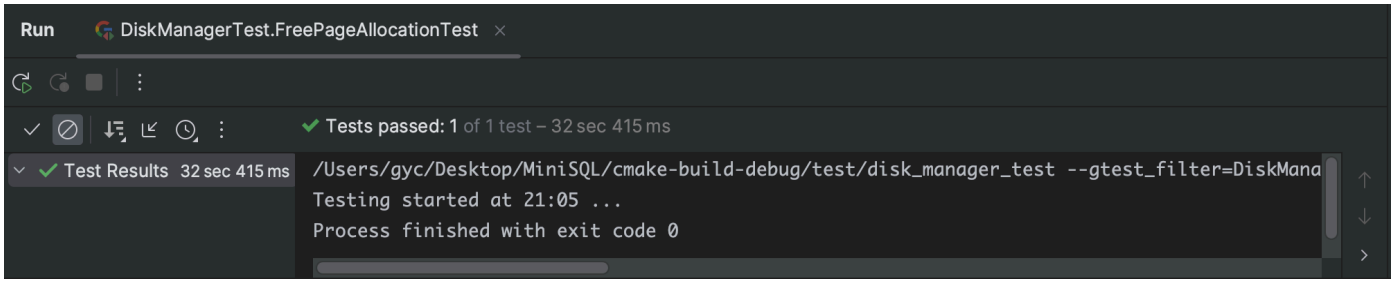
In this top-level module, the implementation tasks seemingly have a common schema that repeatedly appears in each function (👇**Pseudocode**)

```
Page* BufferPoolManager::F(page_id_t page_id) {

  search page_id in page_table_;

  if (/*Found in memory*/) {  // found in memory
    return page_frame;
  }

  // not in memory
  if (/*free_list_ not empty*/) {
    /*do some work to or from free_list_*/;
  }

  //replace
  if(/*none can be replaced, all pinned*/)
    return nullptr;

  update page_table_;

  update metadata of the pageframe&itself;

  load data from disk to memory or reversed;

  return page_frame;
}
```

🖋 Some Notes while Coding

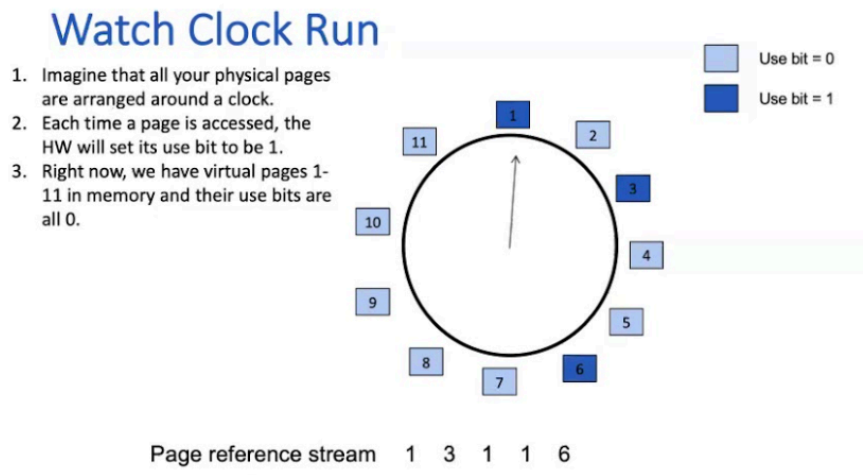
- `free_list_` When a page is removed from the buffer pool, the page frames for that page are added to the free list so that subsequent page allocation operations can select available page frames from it. **And it's still in memory.**
- `frame_id` is within memory while `page_id` is universal overall the whole database system.
- The `page_id` of a page in `free_list_` of the buffer pool is usually `invalid_page_id`. Because having `frame_id` preserved is more than enough for keeping track of it. So when releasing one page we practically set its page id to be invalid.

💻 Test Result:



BONUS: Clock replacemet

The Clock replacement policy is a common&low-cost page replacement algorithm. It is based on the concept of a "clock"(a circular couting mechanism) to determine which page should be replaced.

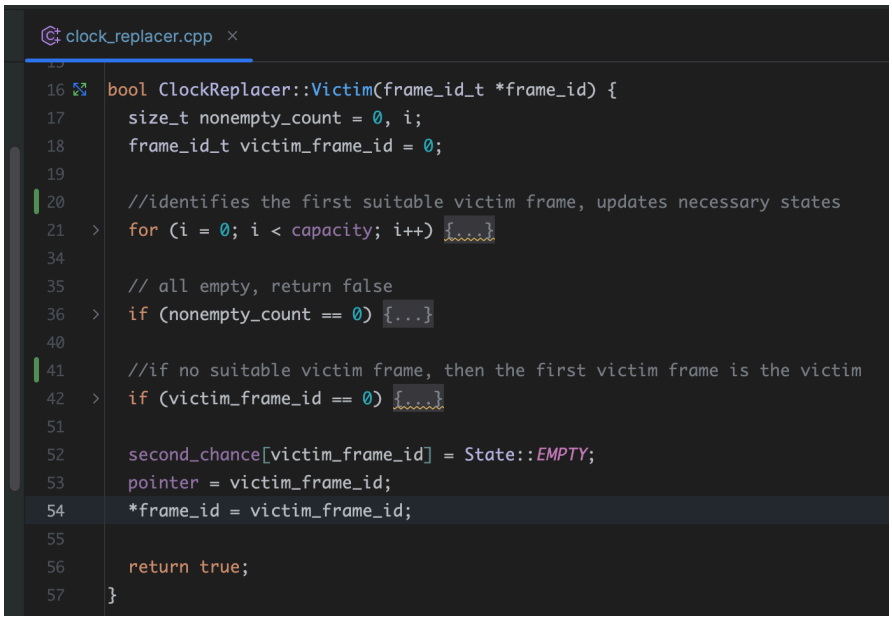


The algorithm maintains a circular linked list where each node represents a page. Each page has a flag called the "reference bit," which indicates whether the page has been recently accessed. (👇pseudocode)

```
// Initialization
Create a circular linked list of pages
Set reference bit of all pages to 0
Set pointer (clock hand) to head of linked list

// Page Replacement
while (true) {
```

```
    if (current_page->reference_bit == 0) {
        // Select current page as replacement candidate
        selected_page = current_page
        current_page->reference_bit = 1
        break
    }
    current_page->reference_bit = 0
    current_page = current_page->next
}

// Replace Selected Page
ReplacePage(selected_page) // Replace selected page with new page to be brought into buffer pool
UpdateMetadata(selected_page) // Update necessary data structures and metadata
```

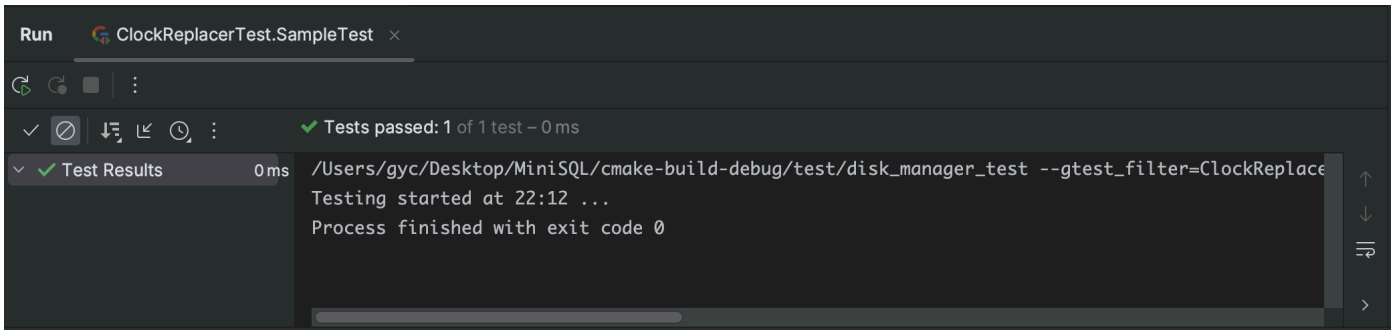
🧪Take `Victim` as example:



👩how the second chance mechanism works：

1. When the Clock replacer starts the victim selection process, it begins scanning the pages in a circular manner, starting from a specific position (the clock hand).
2. For each page encountered during the scan:
    - If the page's reference bit is 0, it means the page has not been recently accessed and can be selected as a victim.
    - If the reference bit is 1, it means the page has been recently accessed and should be given a second chance.
        - The reference bit is then reset to 0, indicating that the page has had its chance.
        - The replacer continues the scan, searching for other suitable victim candidates.
3. If no pages with a reference bit of 0 are found during the initial scan, indicating that all pages have been recently accessed, the replacer performs another scan to find the first page with a reference bit of 0 and selects it as the victim.

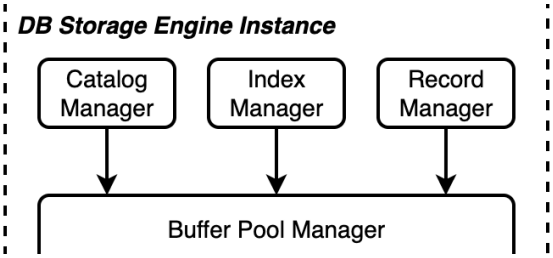
💻 Test Result:



第二模块报告

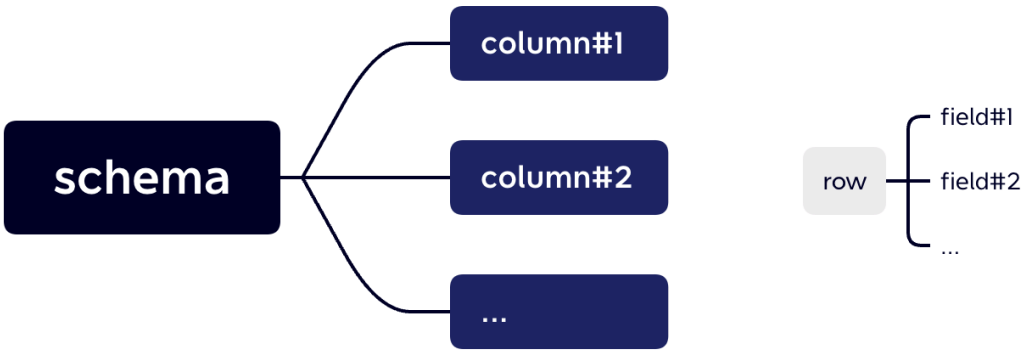
Principles & Theories

Record manager is the part in a DB instance that make requests to buffer pool and receive requests from `executor`.



💡The Executor will send a request to the Record Manager to get the designated record or perform a proper action. Based on the request of the Executor, the Record Manager will read, update, or delete records and return the results to the Executor.
💡When the Record Manager needs to read or write a record, it first checks in the Buffer Pool. If present, the data is read or written directly from the Buffer Pool, avoiding disk access. If they do not exist, the corresponding data page needs to be read from disk into the Buffer Pool.

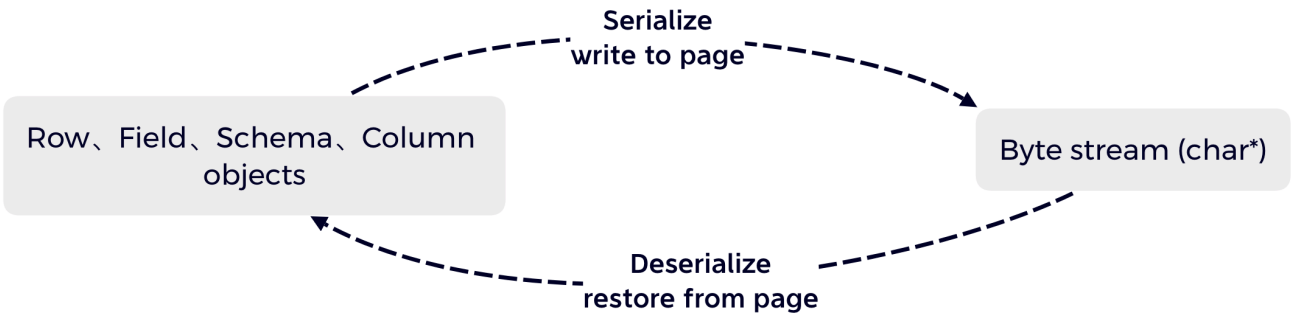
The Record Manager supports record insertion, deletion and lookup operations, and provide the corresponding interface for other adjacent modules.



The `Field` stores information of a particular field in one record, such as the data type , whether it is empty, the value stored and so on, while `column` stores the common constraints about all fields of this column.

A Warm-up: Serialization & Deserialization

Serialization and deserialization operations are actually the process of converting the objects (including records, indexes, directories, etc.) in the database system to the format of internal and external memory. The former converts the logical data (i.e., objects) in memory into physical data(serialized data) that is convenient for storing in files in a certain way, and the latter recovers the logical data from the stored physical data.



For the sample pseudocode,

In the SerializeA function, the id, length of the name and the string itself are written into the buffer serially.

In the DeserializeA function, read 4 bytes from buf, write to id, and push buf back 4 bytes, then do the same to len and char*.

🪄 MAGIC_NUM, written to the head of the byte stream during serialization and read out during deserialization to verify that the object we generated is as expected.

📓 TASK: In this section, we need to complete the SerializeTo, DeserializeFrom, and GetSerializedSize methods for `Row`, `Schema`, and `Column` classes. `Field` methods are given as examples.

Now let`s see into class definitions.

For `Column`, We can use `MACH_WRITE_TO(Type, buf, DATA)` to serialize all members but `string`, for which we write the number of bytes of the string first and then serialize them with `memcpy`. Take `SerializeTo` of the `Column` as an example.

```
uint32_t Column::SerializeTo(char *buf) const {
  uint32_t ofs = 0, len;

  MACH_WRITE_TO(uint32_t, buf, COLUMN_MAGIC_NUM);
  ofs += sizeof(uint32_t);
```

```
  len = name_.size() * sizeof(char);  // bytes of string
  memcpy(buf + ofs, &len, sizeof(uint32_t));
  ofs += sizeof(uint32_t);
  memcpy(buf + ofs, name_.c_str(), len);
  ofs += len;

  /*other metadata`s processing*/
  return ofs;
}
```

‼️ Something should pay attetion to:

- For serialization of `Row` objects, `null` fields can be marked as Bitmaps (i.e., Null Bitmaps).
- For deserializing each field, the memory of the deserialized is maintained by the `Row` object.

Managing records with TableHeap

RowId

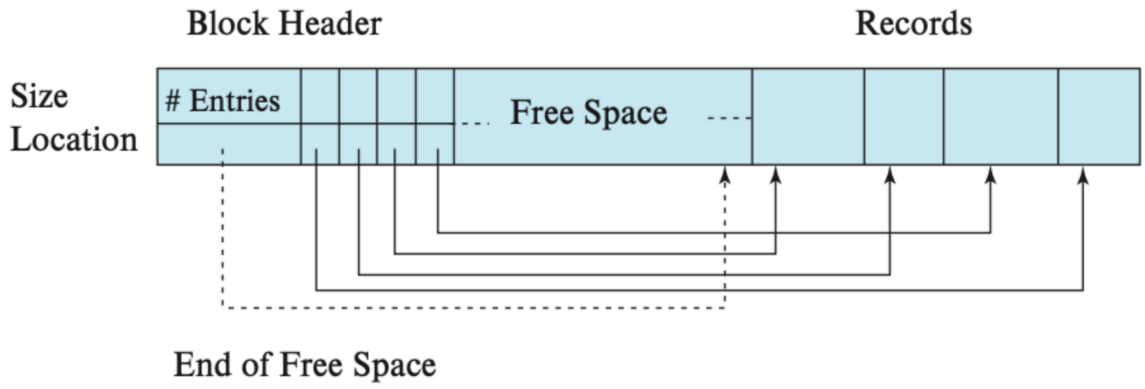
A RowId act like a unique identifier to a row.

- (In the physical sense,) it is a 64-bit integer that uniquely identifies each row of records.
- (In a logical sense,) the top 32 bits store the page_id of the page in which the row is located, and the bottom 32 bits store the row in the page in which the row is located.

TableHeap

TableHeap is a data structure that organizes records in the form of an unordered heap, and different tablepages are connected by a doubly linked list.

The structure of a `table_page`:



⚠️ The header extends from left to right in the page while the inserted records expands from right to left in the page.

insert

When a record is inserted into a heap table, a simple way to do this is to sequentially search through a linked list of tablepages until you find the First TablePage that can hold the record (the First Fit strategy).

delete

When a row with a given RowId needs to be deleted from the heap table, the framework provides a way to physically Delete the row by marking it as deleted with a `Delete Mask` and then physically deleting it at a later time.

update

If enough room in the page, updated directly in the data page. If not, create a new page. 🔦The return arguments of UpdateTuple should be modified to reflect different error conditions

Implementation of TableHeap

`TableHeap` consists of three parts: table head, free space and inserted data. 3 functions to be implemented 👇.

- `bool InsertTuple(Row &row, Transaction *txn)`
- `bool UpdateTuple(const Row &row, const RowId &rid, Transaction *txn)`
- `void ApplyDelete(const RowId &rid, Transaction *txn)`

`InsertTuple`

Deploy the First Fit strategy.

1. **Transaction handling**: At the start, there is a comment stating "transaction not implemented", which implies that this version of the function does not yet incorporate logic for handling transactions.
2. **Serialization size check**: It then gets the serialized size of the `row` object and compares it against the maximum size of a page. If the serialized size of the `row` is larger than the maximum size of a page, the function returns false. This indicates that if the size of a record exceeds the maximum limit of a single page, then this record cannot be inserted in the current page.
3. **Fetching the first available page**: Next, the function tries to fetch the first page of the table heap. This is done by calling `buffer_pool_manager_->FetchPage(first_page_id_)`. If fetching the page fails, the function again returns false. If succeeded, try to insert. The process may be looping for several times before a successful insertion.

`UpdateTuple`

✈️The framework of UpdateTuple

1. **Fetching the Page**: Initially, it tries to fetch the page that contains the record to be updated, by calling `buffer_pool_manager_->FetchPage(rid.GetPageId())`. If fetching the page fails, the function returns false.

2. **Updating the Record**: Next, it creates an `old_row_` object as a copy of the original record, then attempts to update the record on the page using the `page->UpdateTuple(row, &old_row_, schema_, txn, lock_manager_, log_manager_)` method. The return of this method is a `TablePage::RetState` enum, representing the status of the update.

   🔑 RetState enum is a self-defined variable for distinguishing update error types.

3. **Handling Update Status**: Depending on the result of `UpdateTuple`, the function takes different actions:
   - `ILLEGAL_CALL`: If the returned status is illegal call (ILLEGAL_CALL), the function unpin the page with `buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false)`, and returns false.
   - `INSUFFICIENT_TABLE_PAGE`: If the returned status is insufficient table page (INSUFFICIENT_TABLE_PAGE), the function marks the old record for deletion with `MarkDelete(rid, txn)`, then inserts the old record with `InsertTuple(old_row_, txn)`, and finally unpin the old record's page, returning true.
   - `DOUBLE_DELETE`: If the returned status is double delete (DOUBLE_DELETE), the function unpin the page and returns false.
   - `SUCCESS`: If the returned status is success (SUCCESS), the function unpin the page and returns true.

The capability of handling and distinguishing error types is made possible by adding

```
enum class RetState { ILLEGAL_CALL, INSUFFICIENT_TABLE_PAGE,DOUBLE_DELETE,SUCCESS };
```

to table_page.This achieves delicate error handling with only minor changes to the current framework.

`ApplyDelete`

Mark the page as dirty.

`Begin` & `End`

The `Begin` method in the `TableHeap` class retrieves the first valid record in the heap, and returns an iterator to it, while the `End` method returns an iterator representing the end of the table heap, signaled by an invalid page ID. Together, these methods enable iteration through the records in the heap from beginning to end.

TableHeap Iterator

The function retrieves and copies a record from the table page. If the record is deleted, or the provided row object or slot number is invalid, the function will return false. Otherwise, upon successful copying of data from the table page, the function will return true.

`GetTuple`

✈️ Here's a brief walkthrough of the function:

1. **Row validation**: It first asserts the validity of the `row` and its `RowId`.
2. **Slot number retrieval**: It retrieves the current slot number.
3. **Slot number validation**: If the slot number exceeds or equals the count of tuples, the function returns false.
4. **Tuple size retrieval**: It retrieves the size of the current tuple.
5. **Tuple deletion check**: If the tuple is deleted, the function returns false.
6. **Tuple data copying**: After passing all checks, the function copies the tuple data into the `row` object by deserializing the data from the page.
7. **Deserialization validation**: Finally, it asserts that the number of bytes read during deserialization equals the size of the tuple.

`FreeHeap`

The `FreeHeap` function in the `TableHeap` class frees all pages of data in the heap by traversing through them. It retrieves each page by its ID, deletes it from the buffer pool manager, and then moves on to the next page until it encounters a page with an invalid page ID.

operations with iterator

The `++` operator in the `TableIterator` class is implemented to traverse through the rows in a database table.

- In the prefix version, the operator retrieves the next row's ID in the current page, and if it fails, it moves to the next page until a valid row is found or it reaches the end of the table. Then it instantiates a new `Row` object with the ID of the next tuple and updates the current `Row` pointer.
- The postfix version first creates a clone of the current iterator, then increments the original iterator, and finally returns the clone, thereby effectively providing the previous state of the iterator.

Test Results

`tuple_test`

This test program we designed aims to validate the correctness of serialization and deserialization operations for tuples. It includes three test cases: `FieldSDT`, `ColumnSDT`, and `RowSDT`.

The program tests the serialization and deserialization of fields, columns, and rows by creating objects, serializing them into a buffer, deserializing them from the buffer, and performing comparison checks to ensure the accuracy and consistency of the operations, guaranteeing proper data transmission and storage.

⭕ In the build process, I ran into few errors suggesting that

```
/Users/gyc/Desktop/MiniSQL/test/buffer/buffer_pool_manager_test.cpp:14:39:
  std::uniform_int_distribution<char> uniform_dist(0);
                                      ^
1 error generated.
```

It should be written as

```
std::uniform_int_distribution<int> uniform_dist(0);
```

After troubleshooting, the build was done and the results turned out good.

`table_heap_test.cpp`

This test program is designed to validate the functionality of the `TableHeap` class. It performs the following key steps:

1. Sets up the necessary components such as `DBStorageEngine` and `SimpleMemHeap`.
2. Defines a schema using a vector of column definitions.
3. Creates rows with random data and inserts them into the table using `TableHeap`.
4. Verifies that the inserted rows match the expected values.

The purpose of this test is to ensure the correct insertion and retrieval of data using the `TableHeap` class, validating its functionality and reliability in a database system.

💬 When `row_nums` is too large, the bufferpool would be full and all the data pages have been pined.No new data pages can be allocated. If you are to elevate the upper bound, you can change `DEFAULT_BUFFER_POOL_SIZE`.

For a pressure test, I reset the `DEFAULT_BUFFER_POOL_SIZD`

```
15
16  static constexpr int PAGE_SIZE = 4096;                // size of a data page in byte
17  static constexpr int DEFAULT_BUFFER_POOL_SIZE = 4096;// default size of buffer pool
18
```

and with `row_nums` of 2 million. The test took almost 3 minutes but thankfully the result was passed.

第三模块报告

Principles & Theories

📖 Index Manager`s duties:

- index creation and deletion,
- index key equivalence lookup,
- index key range lookup,
- insert and delete key and value operations along with interfaces.

Searching through the heap table to find a row can be very inefficient, and by introducing index we will improve that by a lot. In lab3, we are required to implement a disk-based B+ tree dynamic index structure .

💬 Lookback: What is a B+ tree?

B+ -trees are a common multiway search tree.Each node can have multiple keys and child nodes. Compared with binary search tree, B+ tree can store more key-value pairs in each node, which reduces the height of the tree and improves the search efficiency.

The key/value pairs in the nodes of B+ tree are stored in order of key size, and the leaf nodes form an ordered linked list through Pointers, which makes the range query more efficient. By traversing the entire tree in inorder, an ordered key-value sequence can be obtained.

💬 B+ tree in database systems

Database systems commonly employ B+ trees for indexing to provide efficient data access and query capabilities. For primary Key Indexing, B+ trees are commonly used to uniquely identify each record. They map the primary key values to the physical locations of the records, enabling fast and efficient locating and accessing of specific records.

B+ trees also accelerate data retrieval and access, support range queries and composite indexing, and are suitable for managing indexes in distributed environments.

⚙️ B+ tree reduces the number of disk accesses by maintaining some balance properties, so that the height of the whole tree is kept small.

Implementation of B+ Tree:

The B+ tree is composed of page as basic units, where data are stored. According to the different forms of node data, it can be divided into catagories of `Internal Page` and `Leaf Page`. A `page` can be converted to an `Internal Page` or a `Leaf Page` by type casting.

During the operation, the B+ tree continuously fetches the `Page` from the `buffer pool manager` by `Fetchpage` and `UnpinPage` and mark as dirty to update the B+ tree data on disk.

BPlusTreePage

BPlusTreePage is the common parent class, which contains the data needed by both the intermediate and leaf nodes.

Apart from basic infos, the `lsn_` (log sequence number) of the page plays a key part in crash recovery later labs.

BPlusTreeInternalPage

At any given time, every intermediate node is at least half full by definition.

- When a delete operation results in a node being not half full, it is necessary to merge two adjacent nodes or **redistribute** an element from another node to make it half full.
- When an insertion operation causes a node to overflow the capacity, it needs to be split into two nodes.

This contributes to the most delicate mechanisms of the b+ tree and it`s hard for implementing. Take the`InsertNodeAfter` method as an example:

```
INDEX_TEMPLATE_ARGUMENTS
int B_PLUS_TREE_INTERNAL_PAGE_TYPE::InsertNodeAfter
(const ValueType &old_value, const KeyType &new_key,
const ValueType &new_value)
{
  int old_value_index = this->ValueIndex(old_value);
  std::move_backward(array_+old_value_index+1, array_+GetSize(), array_+GetSize()+1);
  array_[old_value_index+1].first = new_key;
  array_[old_value_index+1].second = new_value;
  IncreaseSize(1);
  return this->GetSize();
}
```

B+ Tree Index Iterator

The B+ index iterator organizes all the leaves into a singly linked list, and then iterates over each key/value pair in the leaf data page in an ordered fashion in a specific direction (this will be comes handy in range queries).

Basically it allows sequential access to the elements, following the order defined by the B+ tree's structure, such as in-order traversal or range-based retrieval. The iterator provides methods to move to the next or previous element and retrieve the current element's key or associated data record.

Tasks:

- Implement the `IndexIterator` of the B+ tree index in `src/include/index/index_iterator.h` and `src/index/index_iterator.cpp`
- Implement the `Begin()` and `End()` functions in the `BPlusTree` class to get the first and tail iterators for the B+ -tree index.

The Design idea of data structure:

- The IndexIterator class uses template parameters to make it generic, allowing it to work with different types of index keys and ValueTypes, as well as different key comparators.
- Private member variables are defined in the class, including the BufferPoolManager pointer (to manage the buffer pool), the Page pointer (to the current page), and the index (the position of the current element in the page).
- The member variables are initialized by the constructor, and the page data pointer is converted to the BPlusTreeLeafPage type for subsequent operations.

📖 Some Notes on Implementation

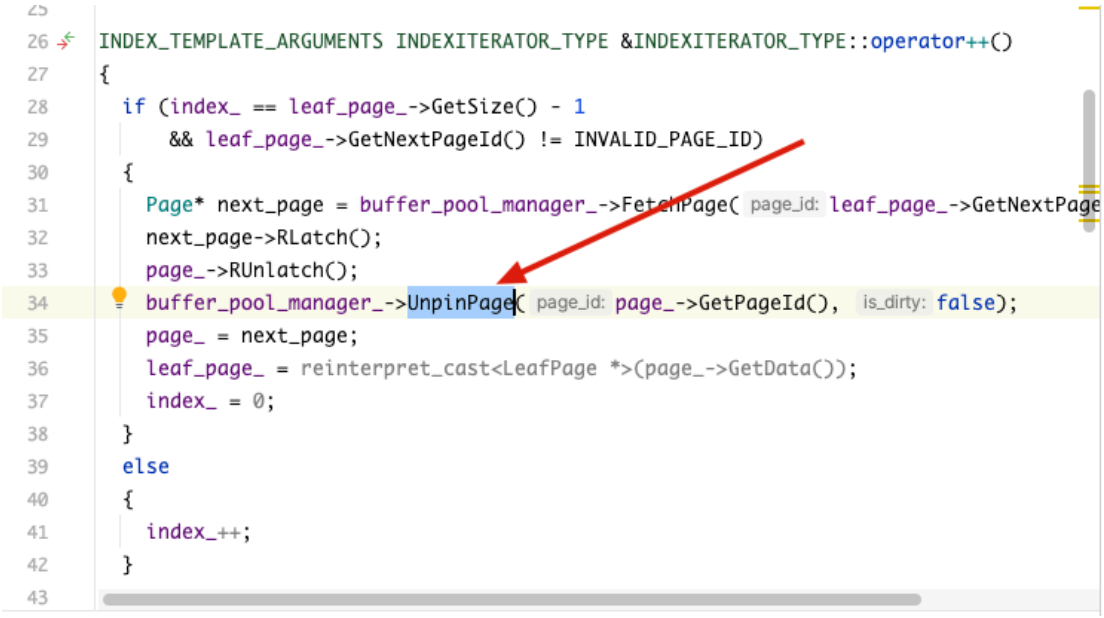
Constructor `IndexIterator`: takes the buffer pool manager, page, and index as arguments and initializes the member variables. At the same time, the page data pointer is converted to the BPlusTreeLeafPage type for subsequent operations.

```
  IndexIterator(BufferPoolManager *buffer_pool_manager,
        Page *page, int index);
```

`operator++()` -Moves the iterator to the next key/value pair If the index of the current page has reached the end of the page and a next page exists, the next page is fetched and the iterator is moved to the first element of the page.

💡 Additionally, pay attention to Iterator lifetime managing. Pin pages in the constructor to ensure that page data is not replaced out of the buffer pool while the iterator is in use. Unlocking (RUnlatch) and unpinning (UnpinPage) in the destructor frees the associated resources.

Here is an example 👇



Test of B+ tree & its index

`b_plus_tree_test`

This test code validates the basic functionality of the B+ tree through insertion, deletion, and lookup operations.

It initializes the storage engine and the B+ tree, prepares test data, shuffles the data, inserts it into the tree, and performs assertions to check the correctness of the operations. It also tests the deletion of keys and verifies the expected results.

💡 To be noted, the test code takes an `int` vector in random order as `value`, the original order as `key`, inserts it into the B+ tree, and then removes some nodes, compares and tests through the original array.

At the same time, the test code will also `Check` to ensure that each `Page` is `Unpin` properly after the B+ tree operation.

💥 The results showed that we`ve designed the b+ tree correctly.

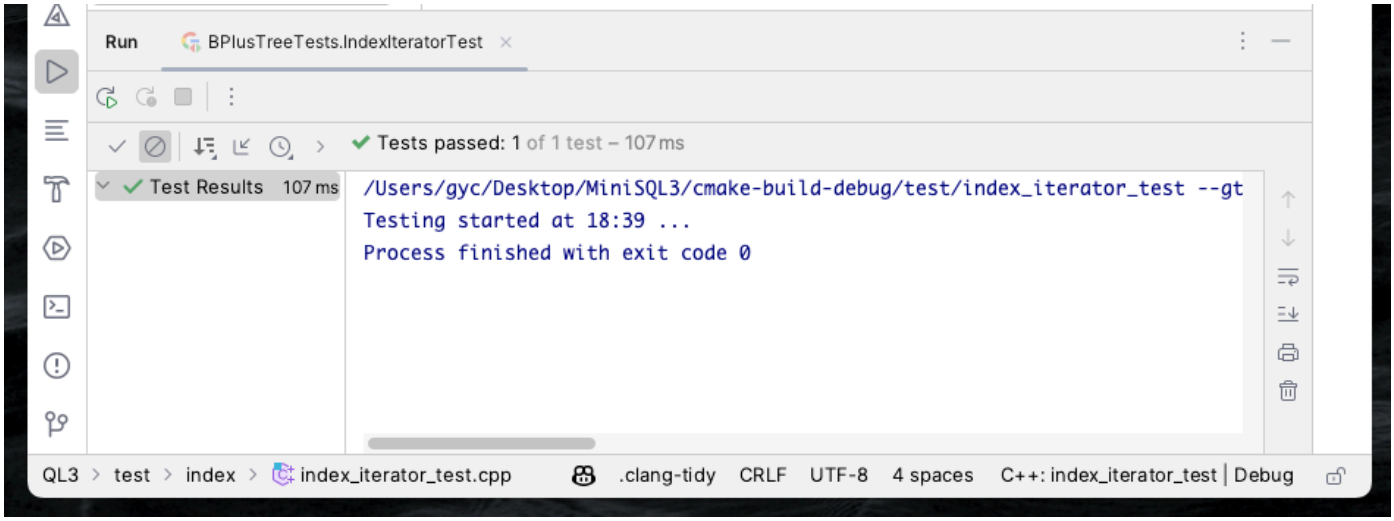
This means that under the given conditions, the system has passed the correctness check and works properly under input situations. Once the correctness of the system has been confirmed, reliability testing can be considered to evaluate the performance of the system under different load and abnormal conditions. So let`s try that by increase the load:

```
14
15 TEST(BPlusTreeTests, SampleTest) {
16   // Init engine
17   DBStorageEngine engine(db_name);
18   BasicComparator<int> comparator;
19   BPlusTree<int, int, BasicComparator<int>> tree( index_id: 0,  bu
20   TreeFileManagers mgr( file_prefix: "tree_");
21   // Prepare data
22   const int n = 100000;
23   vector<int> keys;
24   vector<int> values;
25   vector<int> delete_seq;
26   map<int, int> kv_map;
```

Rerun the test, and it turned out that our B+ tree stays stable.

Test of the iterator

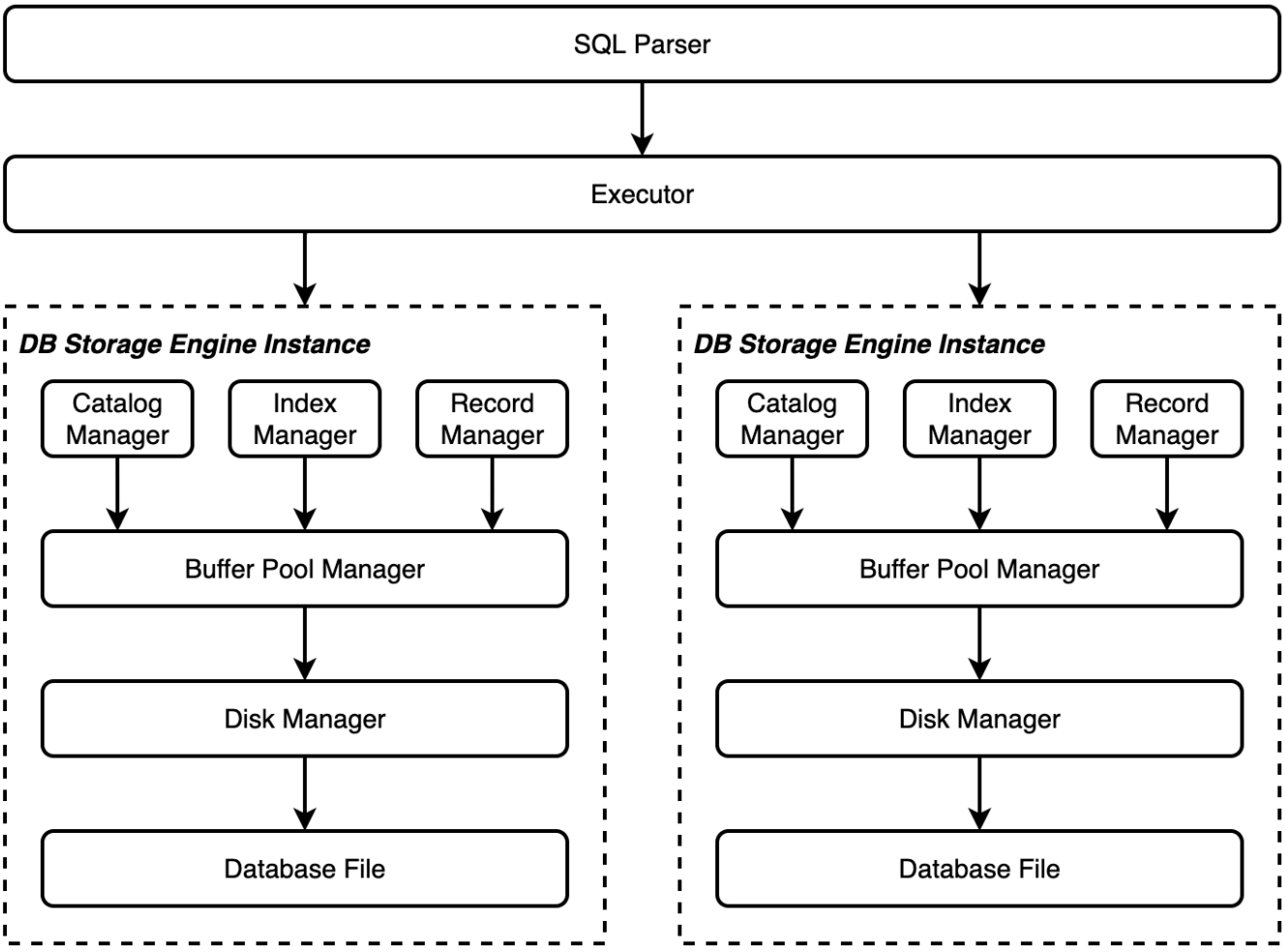
Run the tests, you can see it is a pass.



第四模块报告

Principles & Theories

回看MiniSQL的架构图



Catalog Manager本身是一个对象，包含了index和table的信息，能对具体的table管理，并能为上层的executor提供数据库操作的接口

Implementation of Catalog Manager:

Catalog Manager包含table和index的元信息，需要进行序列话和反序列化以支持长久存取。

元信息

catalog manager:

```
private:
    [[maybe_unused]] BufferPoolManager *buffer_pool_manager_;
    [[maybe_unused]] LockManager *lock_manager_;
    [[maybe_unused]] LogManager *log_manager_;
    CatalogMeta *catalog_meta_;
    std::atomic<table_id_t> next_table_id_;
    std::atomic<index_id_t> next_index_id_;
    // map for tables
    std::unordered_map<std::string, table_id_t> table_names_;
    std::unordered_map<table_id_t, TableInfo *> tables_;
    // map for indexes: table_name->index_name->indexes
    std::unordered_map<std::string, std::unordered_map<std::string, index_id_t>> ind
    std::unordered_map<index_id_t, IndexInfo *> indexes_;
};
```

calalog meta:

```
private:
    static constexpr uint32_t CATALOG_METADATA_MAGIC_NUM = 89849;
    std::map<table_id_t, page_id_t> table_meta_pages_;
    std::map<index_id_t, page_id_t> index_meta_pages_;
};
```

Table info(包括table的meta):

```
private:
    TableMetadata *table_meta_;
    TableHeap *table_heap_;
};
```

Index info(包括index的meta):

```
 private:
  IndexMetadata *meta_data_;
  Index *index_;
  IndexSchema *key_schema_;
};
```

catalog manager不仅包括calalog的meta信息，还包括表名、表、下一个表、索引名、索引、下一个索引等元信息，用于掌控table和index。

序列化与反序列化

实际上metadata的存取都需序列化。我们这里以catalog为例。

```
void CatalogMeta::SerializeTo(char *buf) const {
    ASSERT(GetSerializedSize() <= PAGE_SIZE, "Failed to serialize catalog metadata t
    MACH_WRITE_UINT32(buf, CATALOG_METADATA_MAGIC_NUM);  // magic num
    buf += 4;
    MACH_WRITE_UINT32(buf, table_meta_pages_.size());
    buf += 4;
    MACH_WRITE_UINT32(buf, index_meta_pages_.size());
    buf += 4;
    for (auto iter :pair<...> : table_meta_pages_) {    //page info
        MACH_WRITE_TO(table_id_t, buf, iter.first);
        buf += 4;
        MACH_WRITE_TO(page_id_t, buf, iter.second);
        buf += 4;
    }
    for (auto iter :pair<...> : index_meta_pages_) {    //index
        MACH_WRITE_TO(index_id_t, buf, iter.first);
        buf += 4;
        MACH_WRITE_TO(page_id_t, buf, iter.second);
        buf += 4;
    }
}
```

我们将catalog的metadata化为一个规定的数据结构，首位加入一个硬性规定的magic number用于验证反序列化的data是否合法，然后后面的字段就来自于类的其他成员信息。

反序列化：

```cpp
CatalogMeta *CatalogMeta::DeserializeFrom(char *buf) {
    // check valid
    uint32_t magic_num = MACH_READ_UINT32(buf);  //magic num
    buf += 4;
    ASSERT(magic_num == CATALOG_METADATA_MAGIC_NUM, "Failed to deserialize catalog
    // get table and index nums
    uint32_t table_nums = MACH_READ_UINT32(buf);
    buf += 4;
    uint32_t index_nums = MACH_READ_UINT32(buf);
    buf += 4;
    // create metadata and read value
    CatalogMeta *meta = new CatalogMeta();
    for (uint32_t i = 0; i < table_nums; i++) {
        auto table_id :table_id_t = MACH_READ_FROM(table_id_t, buf);
        buf += 4;
        auto table_heap_page_id :page_id_t = MACH_READ_FROM(page_id_t, buf);
        buf += 4;
        meta->table_meta_pages_.emplace( &: table_id,  &: table_heap_page_id);
    }
    for (uint32_t i = 0; i < index_nums; i++) {
        auto index_id :index_id_t = MACH_READ_FROM(index_id_t, buf);
        buf += 4;
        auto index_page_id :page_id_t = MACH_READ_FROM(page_id_t, buf);
        buf += 4;
        meta->index_meta_pages_.emplace( &: index_id,  &: index_page_id);
    }
    return meta;
```

首先把开头4位取出来，验证这是不是对应的magic number，然后再不停地读取并填入catalog meta的成员，实现读取序列化的data，构造出catalog manager需要的meta。

Catalog Manager

构造：

```
//constructor
CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManage
                               LogManager *log_manager, bool init)
        : buffer_pool_manager_(buffer_pool_manager), lock_manager_(lock_manager),
    //init == true, create a new CM
    if (init) {
        catalog_meta_ = new CatalogMeta;
        next_index_id_.store( d: 0);
        next_table_id_.store( d: 0);
    } else {    //read from old
```

init字段非常关键。当init为true，意思是我们要构造新的catalog，所以把类中的catalog meta等直接调用构造函数即可。如果init为false，意味着我们要从现有的数据读出meta，恢复一个catalog manager。

上层接口

以createtable为例

```
dberr_t CatalogManager::CreateTable(const std::string &table_name, TableSchema *sche
                                    Transaction *txn, TableInfo *&table_info) {
    auto it :iterator<...> = table_names_.find( k: table_name);
    if (it != table_names_.end()) return DB_TABLE_ALREADY_EXIST;
    table_id_t new_table_id_;
```

catalog manager需要给executor提供能调用的接口，比如创表，查表，删表，创索引，删索引等。

以创表为例，catalog manager需要调用下层buffer pool分配出page，有了page后，往上层走，初始化table meta、catalog meta等，并写到磁盘（调用不同的flush函数）。

Test of catalog

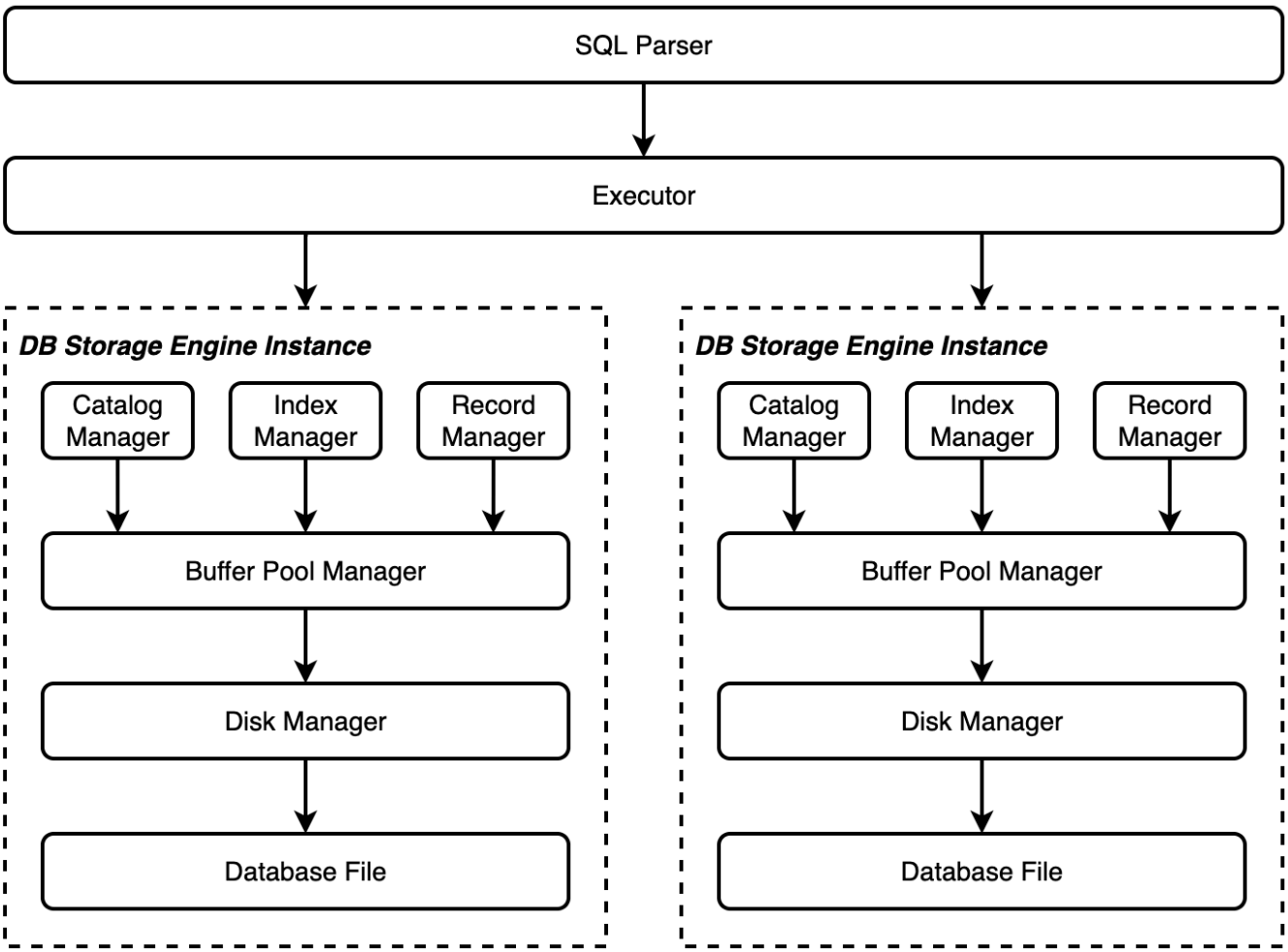
跑测试即可。

```
[----------] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (359 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (339 ms)
[----------] 3 tests from CatalogTest (699 ms total)
```

第五模块报告

Principles & Theories

回看MiniSQL的架构图



Executor之上是parser，parser把生成的语法树交给executor。Executor之下是catalog manager，调用catalog的函数进行对数据库的操作。

Interfaces

在本节中，需要实现seqscan，insert，update，delete, indexscan这五个算子：

- `src/include/executor/executors/abstract_executor.h`
- `src/include/executor/executors/delete_executor.h`
- `src/include/executor/executors/insert_executor.h`
- `src/include/executor/executors/seq_scan_executor.h`
- `src/include/executor/executors/index_scan_executor.h`
- `src/include/executor/executors/values_executor.h`

此外，还需要实现`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：

- `ExecuteEngine::ExecuteCreateDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteDropDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteShowDatabases(*ast, *context)`
- `ExecuteEngine::ExecuteUseDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteShowTables(*ast, *context)`
- `ExecuteEngine::ExecuteCreateTable(*ast, *context)`
- `ExecuteEngine::ExecuteDropTable(*ast, *context)`
- `ExecuteEngine::ExecuteShowIndexes(*ast, *context)`
- `ExecuteEngine::ExecuteCreateIndex(*ast, *context)`
- `ExecuteEngine::ExecuteDropIndex(*ast, *context)`
- `ExecuteEngine::ExecuteExecfile(*ast, *context)`
- `ExecuteEngine::ExecuteQuit(*ast, *context)`

算子实现

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。以顺序扫描算子为例：

init：

```
void SeqScanExecutor::Init() {
    if (!is_init) {
        auto result :dberr_t = exec_ctx_->GetCatalog()->GetTable( table_name: plan_->GetTa
        if (result != DB_SUCCESS) {
            throw MyException( msg: "SeqScanExecutor init failed, table not found");
        }
        table_heap_ = table_info_->GetTableHeap();
        table_iterator_ = table_heap_->Begin( txn: exec_ctx_->GetTransaction());
        is_init = true;
    }
}
```

初始化算子，并且拿到plan想查询的table

next：

```cpp
bool SeqScanExecutor::Next(Row *row, RowId *rid) {
    if (!is_init) {
        throw MyException( msg: "SeqScanExecutor not initialized");
    } else {
        if (table_iterator_ == table_heap_->End()) {
            is_init = false;
            return false;
        } else {
            while (table_iterator_ != table_heap_->End()) {
                *row = *table_iterator_;
                if (plan_->GetPredicate() == nullptr) {
                    Row new_row;
                    row->GetKeyFromRow( schema: table_info_->GetSchema(),  key_schema: p
                    new_row.SetRowId( rid: row->GetRowId());
                    *row = new_row;
                    *rid = row->GetRowId();
                    table_iterator_++;
                    return true;
                }
                auto eva_res :Field  = plan_->GetPredicate()->Evaluate(row);
                if (eva_res.CompareEquals( o: Field( type: kTypeInt,  i: 1)) == CmpBool:

                    Row new_row;
                    row->GetKeyFromRow( schema: table_info_->GetSchema(),  key_schema: p
                    new_row.SetRowId( rid: row->GetRowId());
                    *row = new_row;
```

确定算子初始化后，找db，如果找到，就提取谓词，找满足谓词条件的行。

Engine实现

Execute：

```
dberr_t ExecuteEngine::Execute(pSyntaxNode ast) {
    if (ast == nullptr) {
        return DB_FAILED;
    }
    auto start_time :time_point<...> = std::chrono::system_clock::now();
    unique_ptr<ExecuteContext> context(nullptr);
    if (!current_db_.empty())
        context = dbs_[current_db_]->MakeExecuteContext( txn: nullptr);
    switch (ast->type_) {
        case kNodeCreateDB:
            return ExecuteCreateDatabase(ast,  context: context.get());
        case kNodeDropDB:
            return ExecuteDropDatabase(ast,  context: context.get());
        case kNodeShowDB:
            return ExecuteShowDatabases(ast,  context: context.get());
        case kNodeUseDB:
            return ExecuteUseDatabase(ast,  context: context.get());
        case kNodeShowTables:
            return ExecuteShowTables(ast,  context: context.get());
        case kNodeCreateTable:
            return ExecuteCreateTable(ast,  context: context.get());
        case kNodeDropTable:
            return ExecuteDropTable(ast,  context: context.get());
        case kNodeShowIndexes:
```

接受语法树的节点作为参数，然后解析节点种类，确定调用哪个函数。

函数以droptable作为例子：

```
dberr_t ExecuteEngine::ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context
    clock_t startTime, endTime;
    startTime = clock(); //计时开始
    //TODO: CHECK RIGHT
    ast = ast->child_;
    string database_name = ast->val_;
    auto it :iterator<...> = dbs_.find( k: database_name);
    if (it != dbs_.end()) {
        dbs_.erase( k: database_name);
        remove(("./databases/" + database_name).c_str());
        endTime = clock();
        cout << "Query OK, 1 row affected (" << (double) (endTime - startTime) / C
        return DB_SUCCESS;
    } else {
        cout << "ERROR 1008 (HY000): Can't drop database \'" << database_name << "
        return DB_NOT_EXIST;
    }
}
```

计时开始后，通过语法树节点确定要删的数据库名字是啥，然后层层删除数据库信息。

Test of executor（和其他模块一起走验收流程）

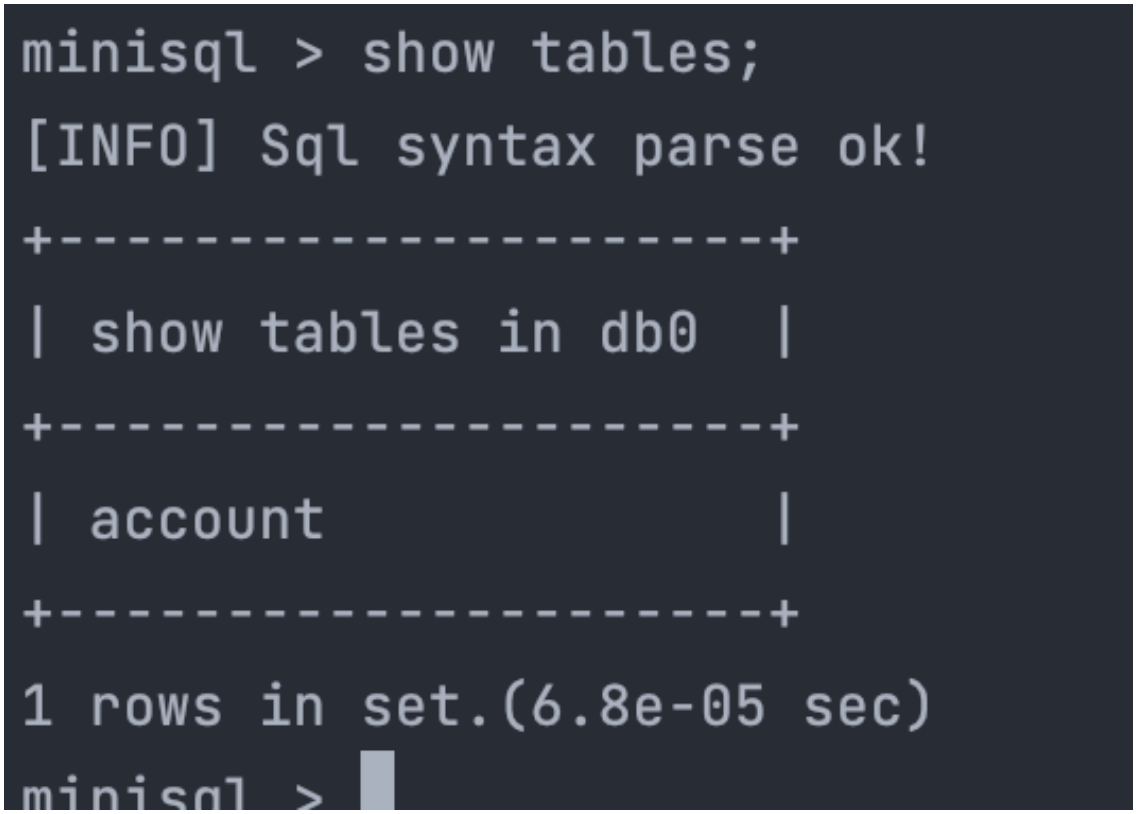
综合测试

创建数据库并插入一万条数据

```
create database db0;
use db0;
create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
insert into account values(12500000, "name00000", 514.35);
insert into account values(12500001, "name00001", 103.14);
insert into account values(12500002, "name00002", 981.86);
insert into account values(12500003, "name00003", 926.51);
insert into account values(12500004, "name00004", 4.87);

insert into account values(12500005, "name00005", 437.08);
```

```
minisql > [INFO] Sql syntax parse ok!
Query OK, 1 row affected(0.0000 sec).
minisql > [INFO] Sql syntax parse ok!
Query OK, 1 row affected(0.0000 sec).
minisql > [INFO] Sql syntax parse ok!
Query OK, 1 row affected(0.0000 sec).
minisql > select * from account;
```

```
| 12509982 | name09982 |
| 12509983 | name09983 |
| 12509984 | name09984 |
| 12509985 | name09985 |
| 12509986 | name09986 |
| 12509987 | name09987 |
| 12509988 | name09988 |
| 12509989 | name09989 |
| 12509990 | name09990 |
| 12509991 | name09991 |
| 12509992 | name09992 |
| 12509993 | name09993 |
| 12509994 | name09994 |
| 12509995 | name09995 |
| 12509996 | name09996 |
| 12509997 | name09997 |
| 12509998 | name09998 |
| 12509999 | name09999 |
+----------+-----------+
10000 row in set(0.1200 sec).
minisql >
```



总结：创数据库创表成功，插入成功，select成功，投影成功

where子句

```
| 12509854 | name09854 | 997.390015 |
| 12509867 | name09867 | 939.140015 |
| 12509875 | name09875 | 936.820007 |
| 12509898 | name09898 | 900.049988 |
| 12509919 | name09919 | 982.640015 |
| 12509926 | name09926 | 927.750000 |
| 12509927 | name09927 | 952.169983 |
| 12509928 | name09928 | 998.280029 |
| 12509934 | name09934 | 917.559998 |
| 12509939 | name09939 | 969.619995 |
| 12509949 | name09949 | 916.770020 |
| 12509950 | name09950 | 921.450012 |
| 12509955 | name09955 | 919.080017 |
| 12509972 | name09972 | 929.669983 |
| 12509976 | name09976 | 974.969971 |
| 12509980 | name09980 | 960.229980 |
| 12509981 | name09981 | 905.479980 |
+----------+-----------+------------+
967 row in set(0.0620 sec).
minisql >
```

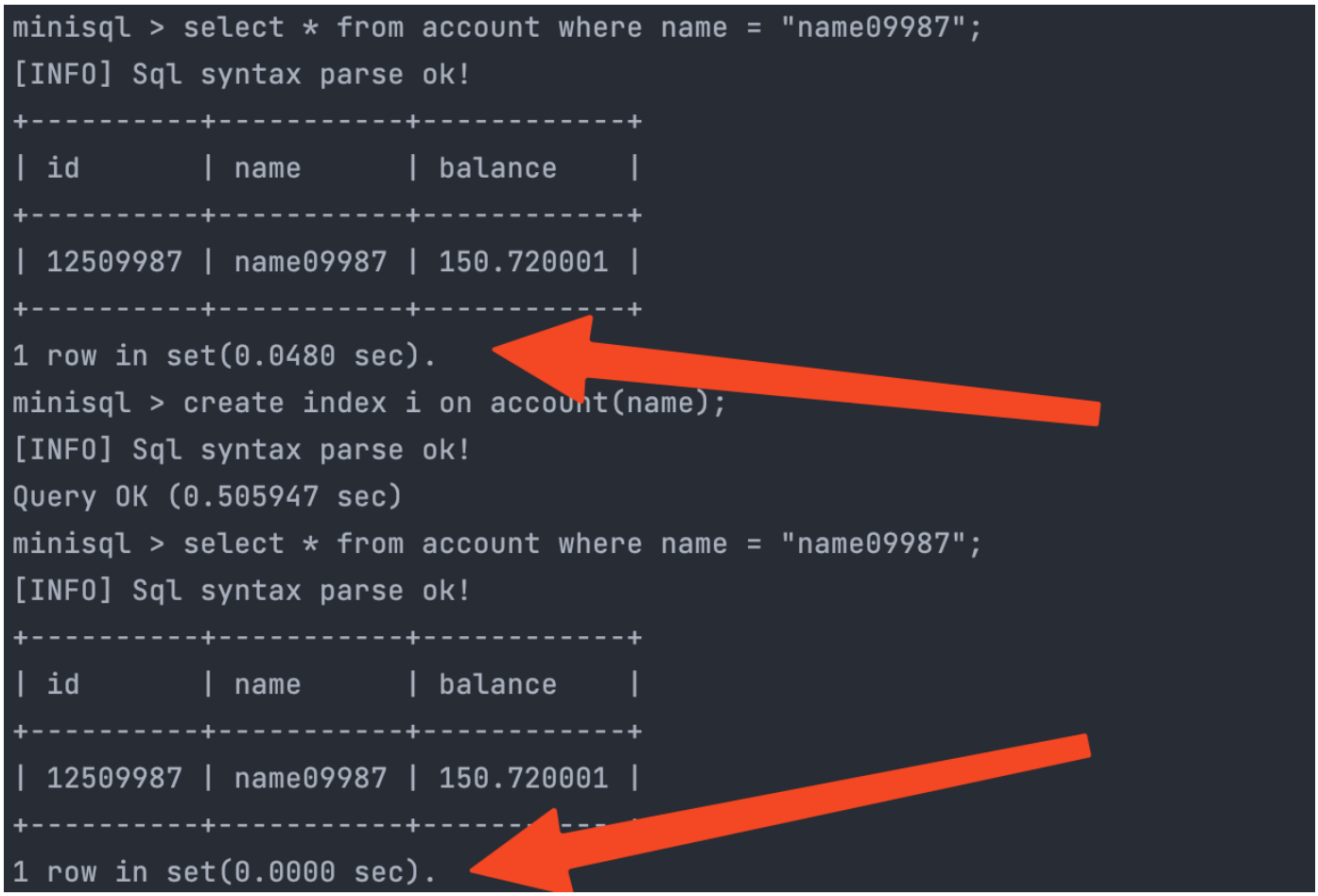
找balance大于900的，成功

索引检验

加索引前找name

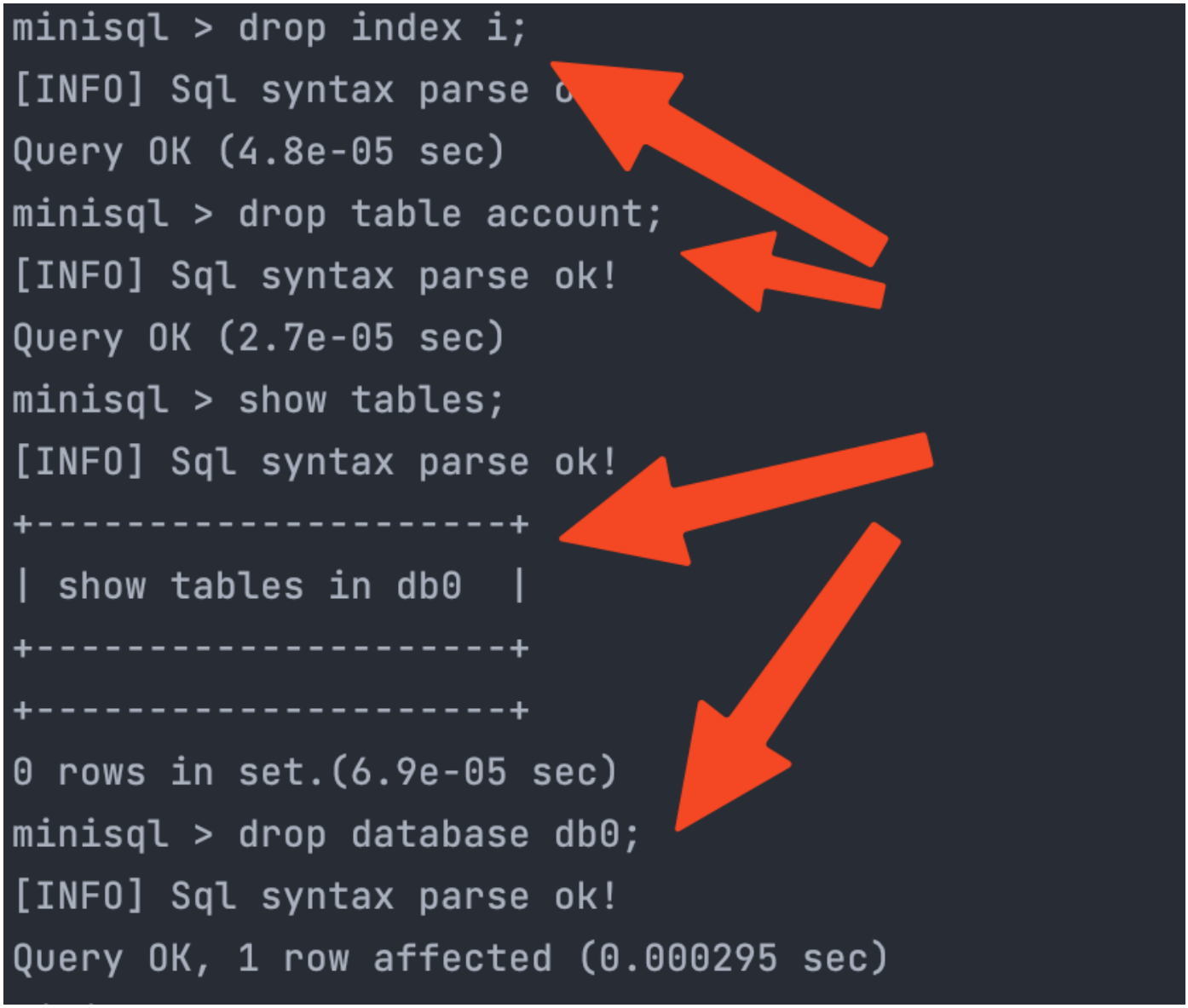
```
minisql > select * from account where name = "name09987";
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12509987 | name09987 | 150.720001 |
+----------+-----------+------------+
1 row in set(0.0480 sec).
```

在name上创索引并查找



可以看出，时间大幅度减少

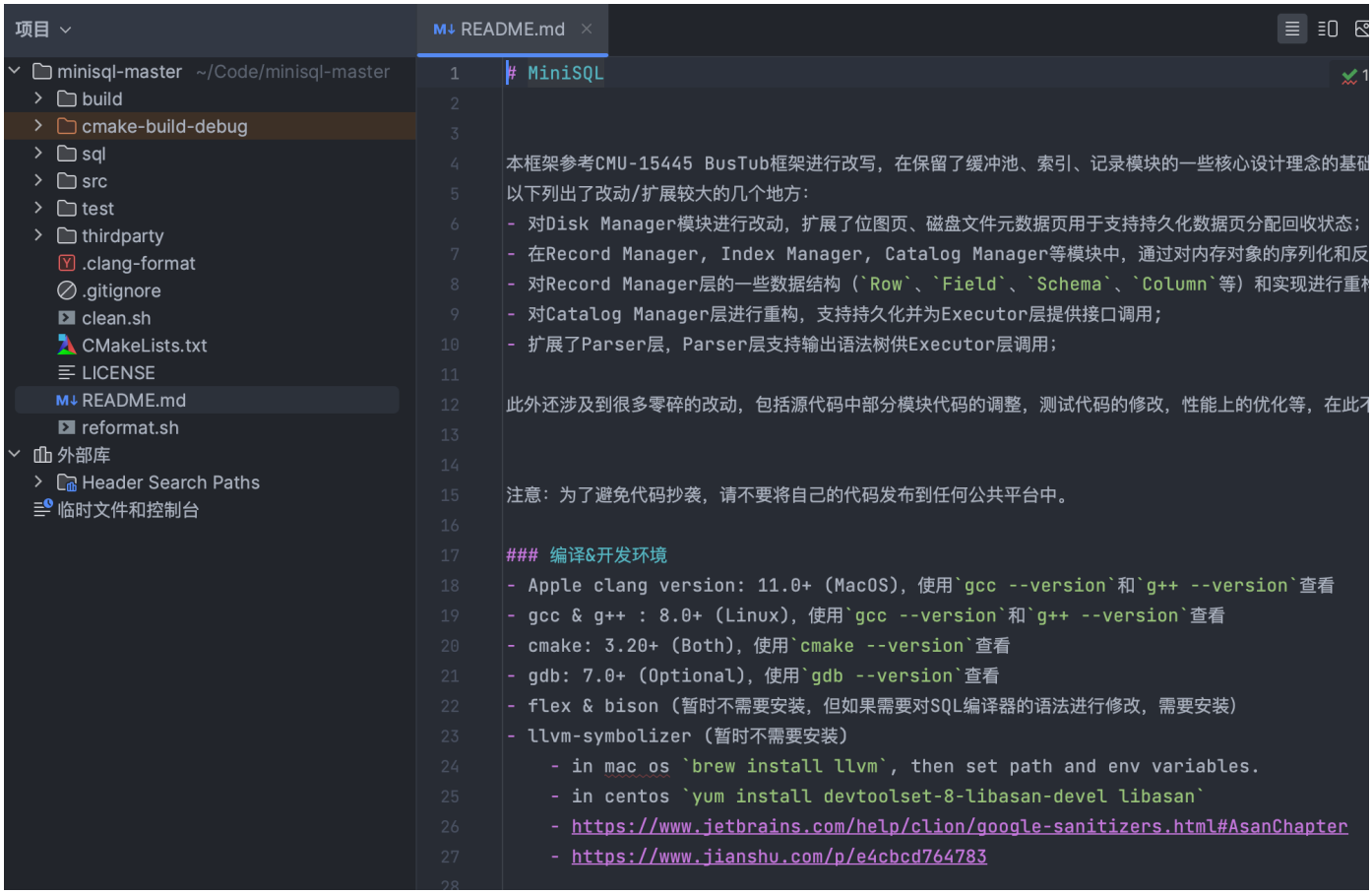
删索引删表删库（删表同时也会删索引，线下验证过）



文件结构

文件夹里code为代码，pdf为报告

代码结构：



build为使用cmake构建的文件夹

cmake-build-debug是CLion调试使用的文件夹

sql是框架验收使用的sql语句

src是源代码

test是test的源代码

thirdparty是依赖包，包括谷歌的gtest

剩余配置文件和README