minisql excutor 个人报告

3200105515 颜晗

本次数据库project中，本人负责实现数据库执行器部分。基本思想即遍历得到的语法树，获取必要的信息和数据，根据信息数据获取对应数据库与表，再从底层到顶层依次构建变量并最终使用下面模块提供的函数进行插入删除与更新。下面将依次对各功能进行讲解，代码部分因为并没有什么特色，而且比较冗余庞大，此处不进行展示。

一、功能分析

1. **语法树**

语法树最高层用于判断操作类型；向下一层用于分析各种条件，例如标记使用数据库（表）的名字的节点，标记需要显示属性名的节点，标记判断条件；再向下一层为显示属性（全部的话在上一层即可判断）和条件（连接符和比较操作符）的具体值。整体按语序进行排列，我们按照形状遍历即可。

2. **构造函数与退出**

执行器其实就是多个数据库引擎的一个`map`容器，所以其构造就是再次调用数据库的构造函数，重点在于修改退出机制，使其在退出时将所有数据库的名称存储于一个文件中，然后就可以在构造中利用名字重新构造数据库，注意重新构造时数据库`init`参数为`false`，因为不能清空原数据。

3. **显示功能与使用数据库**

包括数据库、表、索引在内的所有展示功能都是遍历存容器并去除成员的名字。数据库即遍历执行器内`map`容器，表需要找到当前使用的数据库再遍历，索引由于语法树无法给参数，本人采用遍历并输出数据库所有表对应索引的形式进行输出。

执行器本身有标记当前数据库的字符串变量，使用数据库即遍历所有数据库名，如果有匹配的即数据库存在，将当前数据库字符串修改即可。另外，其余函数在最开始都有判断数据库字符串是否为空（是否有`use database`），防止非法访问出现内存错误，包括各种存在的判断，下面的介绍都略过不谈。

4. **创建与删除**

主要分为数据库、表、索引的创建与删除。

数据库的创建与删除已经有给定的构造与析构函数，只要给定名字即可，注意创建新的数据库与从已有文件恢复数据库引擎的`init`参数是不同的即可。`true`时会导致删除本有的数据库文件建立新的空数据库，`false`会在原有文件的基础上重新打开数据库。

表的创建相对复杂点。得到表的名字后，需要遍历类型节点及其所有分支得到属性的名字和类型，通过属性名与类型先构造`Column`变量存入`vector`容器，再通过该容器构造一个`Schema`变量，声明一个空的表信息指针（`TableInfo*`），即可调用下层的函数进行表的建立。最后有的表声明了主键，而且由于语序排布，主键的声明都在最后，我们需要先拿到所有的声明，还要根据主键来给一些表的列`unique`的属性才能开始属性的构建（因为沿用现成的函数，很少进行增加和修改，部分类的成员一旦构造无法修改），包括得到属性后还需要创建主键的索引，该操作在下面索引部分详细介绍步骤。表的删除只需要获取表的名称即可调用下层的函数进行删除了。

索引建立是表的简化版，拿到表的名字和索引名字，再拿到建立索引的列名，判断`unique`后构建索引的`Schema`(和表的`Schema`是同一类)，调用底层函数即可实现建立。删除同样是通过名字调用底层函数即可，只是由于没有表的名字，需要遍历数据库的所有表。

5. **插入**

插入主要是`Field`以及`Row`的数据获取和构造，调用下层函数即可将`Row`插入对应表中，至于写入硬盘是底层维护的事。另外，我们需要针对`primary key`以及`unique`进行查重，具体方法为通过构造值进行索引的查找，查到了即插入失败，不能有重复值，未查到即可进行插入。最后插入表成功后还需要将对应值插入索引进行维护。

6. **选择、删除、更新**

这三种操作的主要部分都是条件的获取与判断，当然，选择额外加入了使用索引的判断与搜索操作。

选择需要先获取显示的列，后面对表的`Schema`进行遍历对应输出。更新需要获取更新的属性和值，用于后面调用函数。

而对于条件的使用，我们构造了一个新类`cmpData`包括了`Field`类以及一个`string`成员存储对应的比较操作符还有一个用于标记对应值需要比较的列在表中所处位置的"索引"（此索引非上面的索引）成员，而且实现了额外函数对于一组条件三个节点（操作符、属性名、属性值）进行构造变量。另外对于连接符导致的多种条件，我们使用一个`vector`容器来存储一组条件（即使用`and`进行连接，需要同时满足），再使用`vector<vector>`来存储多组（即使用`or`连接）条件，遍历时满足一组条件即可。由于语法树按语序进行构造，多组条件的实现比较麻烦，目前只实现了单个条件或连接符的操作。

具体查找过程，删除和更新都是通过迭代器对表进行遍历一一比较。而查找除此之外还有索引的使用，而由于框架的限制，目前只支持单个条件的相等查找，经过验证可以看出索引的查找效率确实远高于遍历。而对于索引的查找需要构造索引包含的属性对应`Row`然后调用函数进行查找，如果传入的`vectror`中存了有效的`RowId`说明查到了，接下来再利用`RowId`对表进行查找即可大大提升速度。

7. **执行文件**

通过对主函数的过程进行复制，我们便实现了文件执行，只要使用C++的`stream`便可较好复刻过程。只是注意判断文件的结束。

二、个人感想

`minisql`的实现就此结束，从一开始无从下手到最终实现了基本功能，感觉自己还是收获了很多，虽然没有亲自实现底层，但是由于执行器的实现离不开底层各种类与函数，不得不将整个框架与代码看了几遍。另外，`minisql`的复杂对于代码能力的提升也非常明显，对C++的各种功能特性的使用能力都得到了锻炼。

当然目前整个框架仍旧存在一些问题，包括部分提供的代码注释不足，容易导致学生理解偏差进而实现与理想有差距导致出现难以理解的错误，另外，代码中也有一些错漏？比如比较让人难以理解的为什么`TypeChar`类型有`GetData()`函数而另外两个没有实现，而且作为继承却能调用基类的……这种框架本身的代码一般都是通过`.h`文件看功能调用，一旦出现问题比较难查出。总之还是希望未来框架的代码和注释文档能够逐渐完善，减少学生无意义`debug`的过程，毕竟数据库本身也不是主要锻炼代码能力的课程。

MiniSQL小组设计报告

叶瑶波 3200103936 颜晗 3200105515 熊儒海 3200105528

一、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。

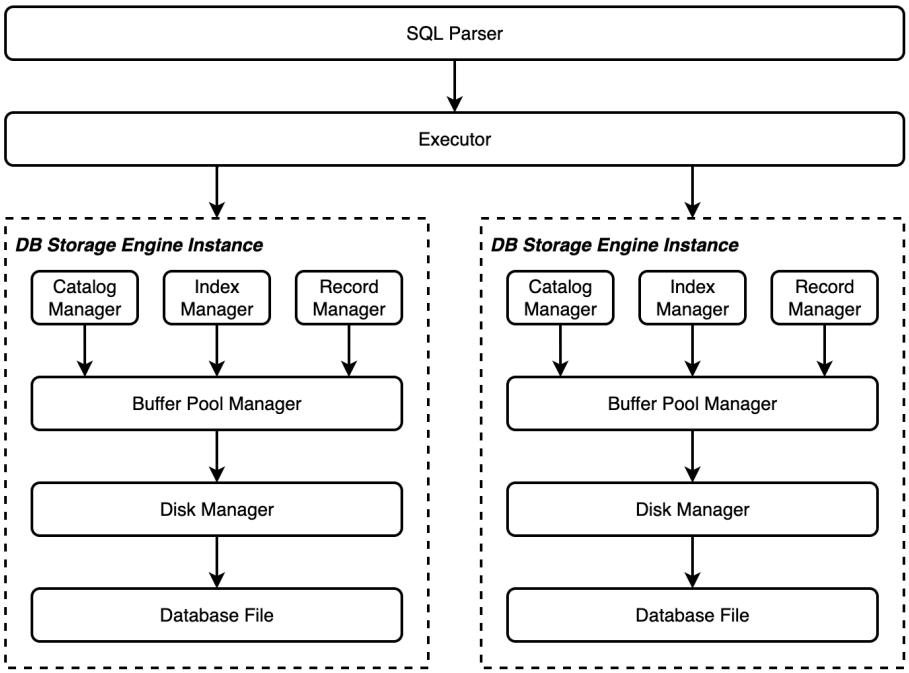
二、实验平台

1. 开发语言：C++
2. 集成开发环境：VSCode
3. 操作系统：WSL:Ubuntu-20.04
4. 源代码管理工具：Git

三、实验内容和要求

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。

四、项目实现(含小组分工)



系统架构如上图，在本次设计中小组分工如下：

叶瑶波：模块1，2，4，即**DISK AND BUFFER POOL MANAGER**，**RECORD MANAGER**，**CATALOG MANAGER**

熊儒海：模块3，即**INDEX MANAGER**

颜晗：模块5，即**SQL EXECUTOR**

4.1 DISK AND BUFFER POOL MANAGER

4.1.1 Disk Manager

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

位图页

位图页由两部分组成，一部分是用于加速Bitmap内部查找的元信息（Bitmap Page Meta），包含当前已经分配的页的数量（`page_allocated_`）以及下一个空闲的数据页(`next_free_page_`)。另一部分是位图存储的具体数据，位图中每个**比特**（Bit）对应一个数据页的分配情况，用于标记该数据页是否空闲（`0`表示空闲，`1`表示已分配)。整个位图页的大小固定为`PAGE_SIZE`（4KB）。

位图页成员变量如下:

```
uint32_t page_allocated_ = 0;           //已经分配的页数
uint32_t next_free_page_ = 0;           //下一个空闲数据页的bit_index
unsigned char bytes[MAX_CHARS] = {0};   //存Bitmap Content
```

实现函数如下:

- `BitmapPage::AllocatePage(&page_offset)`：分配一个空闲页，并通过`page_offset`返回所分配的空闲页位于该段中的下标（从`0`开始）；
    - 先判断已分配页数是否达到上限，如果已满，则分配失败。有空闲页时，将`next_free_page_`赋给`page_offset`，将该位置1。同时要更新元信息，即分配页数加1并找到下一个空闲页的位下标。若分配后达到上限，则将下一空闲页置成无效。
- `BitmapPage::DeAllocatePage(page_offset)`：回收已经被分配的页；
    - 先判断该页是否空闲，如果空闲，则不予回收。否则就将对应的位置0，并更新元数据，同时利用本次回收将`page_offset`的值赋给`next_free_page_`。
- `BitmapPage::IsPageFree(page_offset)`：判断给定的页是否是空闲（未分配）的。
    - 将`page_offset`转化成对应的字节下标`byte_index`和位下标`bit_index`，利用`std::bitset<8>`的`test`函数判断该位是否为1，为1则不空闲，否则返回`true`。

磁盘数据页管理

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

通过增加一层元数据，来管理(位图页+数据页)的组合单元，即一个分区。进行一波套娃，增加磁盘文件能够维护的数据页信息。套娃是无止境的:) Disk Meta Page是数据库文件中的第`0`个数据页，它维护了分区相关的信息，如分区的数量、每个分区中已经分配的页的数量等等。接下来，每一个分区都包含了一个位图页和一段连续的数据页。

但是由于元数据页并不存储数据库数据，但又在物理上占用了数据库文件中的数据页，导致真正存储数据的数据页不连续。

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

我们希望上层的Buffer Pool Manager对于Disk Manager中的页分配无感知，即对Buffer Pool Manager来说，连续分配得到的页号是连续的（0, 1, 2, 3...）。为此，在Disk Manager中，需要对物理页号做一个映射（映射成上表中的逻辑页号）。

相关成员变量如下：

```
char meta_data_[PAGE_SIZE] = {0};  //是MetaPage在内存中的缓存
```

实现函数如下：

- `DiskManager::AllocatePage()`：从磁盘中分配一个空闲页，并返回空闲页的**逻辑页号**；
  - 先判断是否有可供分配的数据页，若没有，则返回`INVALID_PAGE_ID`。如果有，则找到第一个有空闲页的分区，如果在已分配的分区内没有空闲页，则分配新分区。找到分区后计算该分区位图页对应的物理页号，然后从磁盘读取位图页。利用位图页信息分配空闲页，得到相对页号`page_offset`，然后根据映射关系换算成绝对的逻辑页号。完成后更新磁盘管理器元信息，并将位图页信息更新到磁盘中。实际上可以有一个**位图页缓冲区**，一开始读取位图页至缓冲区，然后只需要更新缓冲区信息，最后将脏的缓冲区部分写回磁盘即可，应该可以减少磁盘访存cost。
- `DiskManager::DeAllocatePage(logical_page_id)`：释放磁盘中**逻辑页号**对应的物理页。
  - 先找到该逻辑页号对应的分区，计算其在分区中的相对页号`page_offset`，并计算该分区位图页对应的物理页号，从磁盘读取位图页。利用位图页信息释放该页，释放成功则更新磁盘管理器元信息，并将位图页信息更新至磁盘。
- `DiskManager::IsPageFree(logical_page_id)`：判断该**逻辑页号**对应的数据页是否空闲。
  - 同理，计算逻辑页在分区中的相对页号`page_offset`，读取位图页信息，根据信息判断是否空闲。
- `DiskManager::MapPageId(logical_page_id)`：可根据需要实现。在`DiskManager`类的私有成员中，该函数可以用于将逻辑页号转换成物理页号。
  - 根据映射关系计算一下即可

LRU替换策略

Buffer Pool Replacer负责跟踪Buffer Pool中数据页的使用情况，并在Buffer Pool没有空闲页时决定替换哪一个数据页。我们实现一个基于LRU替换算法的`LRUReplacer`。`LRUReplacer`的大小默认与Buffer Pool的大小相同。

成员变量如下：

```
list<frame_id_t> lru_list_; //可被替换的数据页列表，最新被固定的页被加到尾部
```

实现函数如下：

- `LRUReplacer::Victim(*frame_id)`：替换（即删除）与所有被跟踪的页相比最近最少被访问的页，将其页帧号（即数据页在Buffer Pool的Page数组中的下标）存储在输出参数`frame_id`中输出并返回`true`，如果当前没有可以替换的元素则返回`false`；
  - 若`lru_list_`的size为0则返回`false`，否则将`lru_list_.front()`给`frame_id`，并将其从列表中移除。

- `LRUReplacer::Pin(frame_id)`：将数据页固定使之不能被`Replacer`替换，即从`lru_list_`中移除该数据页对应的页帧。`Pin`函数应当在一个数据页被Buffer Pool Manager固定时被调用；
  - 将`frame_id`从列表中移除
- `LRUReplacer::Unpin(frame_id)`：将数据页解除固定，放入`lru_list_`中，使之可以在必要时被`Replacer`替换掉。`Unpin`函数应当在一个数据页的引用计数变为`0`时被Buffer Pool Manager调用，使页帧对应的数据页能够在必要时被替换；
  - 先查看是否已在列表中，如果在则不做操作，否则将其加到列表尾部，表示是最新被解除固定的页。
- `LRUReplacer::Size()`：此方法返回当前`LRUReplacer`中能够被替换的数据页的数量。
  - 返回列表`size`

4.1.2 Buffer Pool Manager

Buffer Pool Manager负责从Disk Manager中获取数据页并将它们存储在内存中，并在必要时将脏页面转储到磁盘中（如需要为新的页面腾出空间）。

相关成员变量

```
Page *pages_;           // array of pages
std::unordered_map<page_id_t, frame_id_t> page_table_;   // to keep track of pages
std::list<frame_id_t> free_list_;       //空闲页列表
```

实现函数如下：

- `BufferPoolManager::FetchPage(page_id)`：根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取；
  - 先判断页号是否有效，无效则返回`nullptr`。再判断数据页是否空闲，若空闲则说明数据页数据无效，返回`nullptr`。利用`page_table_`判断是否已在缓冲区，如果在，就固定对应的`frame`，并返回对应页。否则要去磁盘读取，如果此时缓冲区既没有空闲页也没有可替换的页，就返回`nullptr`。优先从自由页列表申请空闲的`frame`，如果没有，再去找可替换的`frame`。拿到`frame`后，如果是脏页，需要先写入磁盘，然后从`page_table_`移除该frame中的旧页，加入新页。更新缓冲区中`pages_[frame]`的元数据，并从磁盘读取`page_id`对应页的数据至缓冲区，返回`&pages_[frame]`。
- `BufferPoolManager::NewPage(&page_id)`：分配一个新的数据页，并将逻辑页号于`page_id`中返回；
  - 若缓冲区既没有空闲页也没有可被替换的页，则无法分配新数据页，返回`nullptr`。分配新页，若分配到的页号无效，则返回`nullptr`。分配成功后需要将该页加入缓冲区，优先从自由页列表申请空闲的`frame`，如果没有，再去找可替换的`frame`。拿到`newframe`后更新`pages_[frame]`的元数据，并利用`pages_[newframe].ResetMemory()`清空原来的数据。在`page_table_`中加入新页，返回`&pages_[frame]`。
- `BufferPoolManager::UnpinPage(page_id, is_dirty)`：取消固定一个数据页；
  - 如果缓冲区没有使用这一页，返回`true`。如果是脏页，先将数据页转储到磁盘。然后更新该页的`pin_count_`，原来为0则不变，否则减1。若更新后`pin_count`为0，说明没有其它固定该页的线程，取消固定成功，更新元数据，否则本线程取消固定成功，但总体上取消固定失败。
- `BufferPoolManager::FlushPage(page_id)`：将数据页转储到磁盘中；
  - 如果没用这一页，返回`true`。否则将缓冲区中的该页写入磁盘。
- `BufferPoolManager::DeletePage(page_id)`：释放一个数据页；

    - 如果没用这一页，返回 `true`。如果此页的 `pin_count` 不为0，说明有线程占用，返回 `false`。否则就可以将其从 `page_table_` 移除，重新设置对应 `pages_[frame]` 的元信息，并将该 `frame` 加回自由页列表，再释放该页。
- `BufferPoolManager::FlushAllPages()`：将所有的页面都转储到磁盘中。
    - 将在缓冲区中使用的数据页写回磁盘

4.2 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。

4.2.1 记录与模式

与记录（Record）相关的概念有以下几个：

- 列（`Column`）：用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等等；
- 模式（`Schema`）：用于表示一个数据表或是一个索引的结构。一个 `Schema` 由一个或多个的 `Column` 构成；
- 域（`Field`）：它对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；
- 行（`Row`）：与元组的概念等价，用于存储记录或索引键，一个 `Row` 由一个或多个 `Field` 构成。

为了能够持久化存储上面提到的 `Row`、`Field`、`Schema` 和 `Column` 对象，我们需要将内存中这些对象的**必要信息**序列化成字节流（`char*`），以写入数据页中。与之相对，为了能够从磁盘中恢复这些对象，我们同样需要从数据页的 `char*` 类型的字节流中反序列化出我们需要的对象。为了确保数据能够正确存储，我们在上述提到的 `Row`、`Schema` 和 `Column` 对象中都引入了魔数 `MAGIC_NUM`，它在序列化时被写入到字节流的头部并在反序列化中被读出以验证我们在反序列化时生成的对象是否符合预期。

实现函数如下：

- `Row::SerializeTo(*buf, *schema)`
    - 如果整个 `Row` 不存在 `fields`，则不需要序列化，否则先将空位图(每个bit标识对应的 `field` 是否为 `null`)序列化，再将各个 `field` 序列化，序列化各个 `field` 时，要先将其 `type` 序列化，以使反序列化时能够正确构造 `field` 对象。
- `Row::DeserializeFrom(*buf, *schema)`
    - 按照序列化的顺序，先将空位图读出来，然后依次反序列化各个 `field`，先根据空位图判断是否为 `null`，并将 `type` 反序列化出来，如此就可以去调用 `Field` 自己的反序列化函数了。
- `Row::GetSerializedSize(*schema)`
    - 返回序列化时所需字节数。
- `Column::SerializeTo(*buf)`
    - 先将魔数序列化，然后依次序列化各个成员。在序列化 `string` 型成员时，要先把它的长度序列化，再序列化其内容。
- `Column::DeserializeFrom(*buf, *&column, *heap)`
    - 根据魔数判断数据是否正确，然后依次反序列化各个对象，并用临时变量暂存，然后根据类型去调用 `Column` 的构造函数，构造时用 `heap` 为其分配空间，方便空间的回收。
- `Column::GetSerializedSize()`

    - 返回序列化时所需字节数。
- `Schema::SerializeTo(*buf)`
    - 序列化魔数和各个列，对于列只需要调用其序列化函数即可
- `Schema::DeserializeFrom(*buf, *&schema, *heap)`
    - 根据魔数检验数据正确性，然后反序列化各个列，调用`Schema`的构造函数，用`heap`为其分配空间。
- `Schema::GetSerializedSize()`
    - 返回序列化时所需字节数。

4.2.2 通过堆表管理记录

RowId

对于数据表中的每一行记录，都有一个唯一标识符`RowId`与之对应。`RowId`同时具有逻辑和物理意义，在物理意义上，它是一个64位整数，是每行记录的唯一标识；而在逻辑意义上，它的高32位存储的是该`RowId`对应记录所在数据页的`page_id`，低32位存储的是该`RowId`在`page_id`对应的数据页中对应的是第几条记录。`RowId`的作用主要体现在两个方面：一是在索引中，叶结点中存储的键值对是索引键`Key`到`RowId`的映射，通过索引键`Key`，沿着索引查找，我们能够得到该索引键对应记录的`RowId`，也就能够在堆表中定位到该记录；二是在堆表中，借助`RowId`中存储的逻辑信息（`page_id`和`slot_num`），可以快速地定位到其对应的记录位于物理文件的哪个位置。

堆表

堆表（`TableHeap`）是一种将记录以无序堆的形式进行组织的数据结构，不同的数据页（`TablePage`）之间通过双向链表连接。堆表中的记录通过`RowId`进行定位。`RowId`记录了该行记录所在的`page_id`和`slot_num`，其中`slot_num`用于定位记录在这个数据页中的下标位置。

堆表中的每个数据页都由表头（Table Page Header）、空闲空间（Free Space）和已经插入的数据（Inserted Tuples）三部分组成，表头在页中从左往右扩展，记录了`PrevPageId`、`NextPageId`、`FreeSpacePointer`以及每条记录在`TablePage`中的偏移和长度；插入的记录在页中从右向左扩展，每次插入记录时会将`FreeSpacePointer`的位置向左移动。

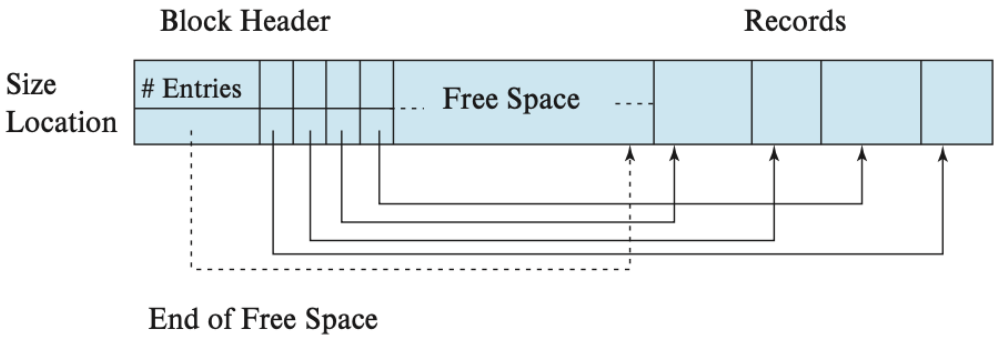


Figure 13.6 Slotted-page structure.

当向堆表中插入一条记录时，一种简单的做法是，沿着`TablePage`构成的链表依次查找，直到找到第一个能够容纳该记录的`TablePage`（*First Fit* 策略）。当需要从堆表中删除指定`RowId`对应的记录时，通过打上Delete Mask来标记记录被逻辑删除，在之后某个时间段再从物理意义上真正删除该记录。此外，在堆表中还需要实现迭代器`TableIterator`以便上层模块遍历堆表中的所有记录。

实现函数如下：

- `TableHeap:InsertTuple(&row, *txn)`：向堆表中插入一条记录，插入记录后生成的`RowId`需要通过`row`对象返回（即`row.rid_`）；

    - 从第一个堆表页开始查找能够容纳该元组的数据页，如果在当前管理的某页中插入成功，则返回 `true`，如果现有的所有堆表页都放不下，就申请新的页，如果申请不到，就返回 `false`。申请到后维护双向链表并初始化新页，将该元组插入到新页中，返回 `true`。
- `TableHeap:UpdateTuple(&new_row, &rid, *txn)`：将 `RowId` 为 `rid` 的记录 `old_row` 替换成新的记录 `new_row`，并将 `new_row` 的 `RowId` 通过 `new_row.rid_` 返回；
    - 根据需要更新的元组的 `RowId` 找到该元组所在堆表页，然后调用 `TablePage::UpdateTuple` 进行更新。如果返回0，说明更新成功；如果返回-1说明该 `slot num` 无效或者该元组已被删除，更新失败；如果返回1说明空间不足，无法在原位置进行更新，此时需要执行删除/插入操作，先将该记录标记删除，然后插入更新后的记录，如果插入成功则返回 `true`，如果插入失败，需要回滚删除操作，并返回 `false`。
- `TableHeap:ApplyDelete(&rid, *txn)`：从物理意义上删除这条记录；
    - 根据 `rid` 找到该记录所在页和slot，调用 `TablePage::ApplyDelete`，从物理意义上删除
- `TableHeap:GetTuple(*row, *txn)`：获取 `RowId` 为 `row->rid_` 的记录；
    - 根据 `row` 中的 `RowId` 信息调用 `TablePage::GetTuple` 获得记录
- `TableHeap:FreeHeap()`：销毁整个 `TableHeap` 并释放这些数据页；
    - 从第一页开始，遍历堆表中的堆表页链表，依次释放各个页，释放完成后将 `first_page_id_` 置成无效。
- `TableHeap::Begin()`：获取堆表的首迭代器；
    - 获得堆表页中第一个有效元组的 `RowId`，如果获取成功，则用该 `RowId` 构造迭代器，作为首迭代器返回。如果获取失败，则用无效的 `RowId(INVALID_PAGE_ID, 0)` 构造迭代器并返回。
- `TableHeap::End()`：获取堆表的尾迭代器；
    - 用无效的 `RowId(INVALID_PAGE_ID, 0)` 构造迭代器并返回。
- `TableIterator::operator++()`：移动到下一条记录，通过 `++iter` 调用；
    - 在同一页中获取下一个元组的 `RowId`，获取到后更新迭代器中 `iter` 的 `RowId`，如果到达页尾，就开始遍历下一页继续寻找。找到后利用 `iter` 的新 `RowId` 从堆表中获得对应元组。如果遍历到了尾迭代器，就保持不变
- `TableIterator::operator++(int)`：移动到下一条记录，通过 `iter++` 调用。
    - 与 `TableIterator::operator++()` 基本一致，只是返回的是++前的迭代器

4.3 INDEX MANAGER

主要实现文件如下

- src/include/page/b_plus_tree_page.h
- src/page/b_plus_tree_page.cpp
- src/include/page/b_plus_tree_internal_page.h
- src/page/b_plus_tree_internal_page.cpp
- src/include/page/b_plus_tree_leaf_page.h
- src/page/b_plus_tree_leaf_page.cpp
- src/include/index/b_plus_tree.h
- src/index/b_plus_tree.cpp
- src/include/index/index_iterator.h
- src/index/index_iterator.cpp

插入的实现

最开始就是实现PARENT PAGE，它是INTERNAL PAGE 和 LEAF PAGE的父类。抽出了2个PAGE的共有变量和方法。难点是如何实现GetMinsize

```
int BPlusTreePage::GetMinSize() const {
  if(IsRootPage())
    return IsLeafPage()?1:2;
  else
    return (max_size_+1)/2;
}
```

在这里，应首先判断是不是根节点，而根节点的判断又应分为是叶节点或不是叶节点，因为如果根不是叶节点那么，至少应当有两个孩子，最小size应是2，而是叶节点则应为空树了。

下面是实现，INTERNAL PAGE，也就是B+树的非叶子节点，存储的是ROUTE 信息。实现其中基本的函数。之后实现，LEAFPAGE，也就是B+树的叶子节点，存储的是数据信息。同样实现其中基本的函数。

之后在B tree中实现必要的查找函数，为插入做准备

```
function find(value V)
/* Returns leaf node C and index i such that C.P_i points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that V ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (V = C.K_i)
            then Set C = C.P_{i+1}
        else C = C.P_i /* V < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i
        then return (C, i)
        else return null ; /* No record with key value V exists*/
```

按照上述伪码实现查找函数。

实现必要的函数之后，接着就是实现插入的功能。

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L'
            Copy L.P_1 ... L.K_{n−1} to a block of memory T that can
                hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L'.P_n = L.P_n; Set L.P_n = L'
            Erase L.P_1 through L.K_{n−1} from L
            Copy T.P_1 through T.K_{⌈n/2⌉} from T into L starting at L.P_1
            Copy T.P_{⌈n/2⌉+1} through T.K_n from T into L' starting at L'.P_1
            Let K' be the smallest key-value in L'
            insert_in_parent(L, K', L')
        end
```

```
procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K1)
        then insert P, K into L just before L.P1
        else begin
            Let Ki be the highest value in L that is less than K
            Insert P, K into L just after T.Ki
        end

procedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'    /* N and N' are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P1 ... T.P⌈n/2⌉ into P
            Let K'' = T.K⌈n/2⌉
            Copy T.P⌈n/2⌉+1 ... T.Pn+1 into P'
            insert_in_parent(P, K'', P')
        end
```

Figure 11.15 Insertion of entry in a B+-tree.

按照上述伪码，主要实现的是Insert,insert into leaf,insert into parent,这三个函数，以及其中需要调用的子函数。其中Insert作为顶层函数，实现方式如下

```
INDEX_TEMPLATE_ARGUMENTS
bool BPLUSTREE_TYPE::Insert(const KeyType &key, const ValueType &value, Transaction *transaction) {
  if(IsEmpty()){
    StartNewTree(key,value);
    return true;
  }
  else return InsertIntoLeaf(key,value,transaction);
}
```

在按照要求实现好插入的函数之后，插入的工作基本就完成了。

之后为了完成测试，还需要完成迭代器的实现。

3.2删除操作的实现

伪代码如下

```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
                        pair in N'
                    remove (N'.P_m, N'.K_m) from N'
                    insert (N'.P_m, N'.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_m
                end
            end
            else ... symmetric to the then case ...
        end
    end
```

Figure 11.19 Deletion of entry from a B+-tree.

Remove的思想，是找到对应的LEAF PAGE,然后删除对应的项，然后判断是否需要合并或者重新分配。

```
INDEX_TEMPLATE_ARGUMENTS
void BPLUSTREE_TYPE::Remove(const KeyType &key, Transaction *transaction) {
  //If current tree is empty, return immediately.
  if(IsEmpty()) return;
  //first find the right leaf page
  Page * deLeafpage=FindLeafPage(key);
  LeafPage *deleaf=reinterpret_cast<LeafPage* >(deLeafpage->GetData());

  //delete entry
  int deletesize=deleaf->RemoveAndDeleteRecord(key,comparator_);

  //deal with redistribute or merge
  if(deletesize<deleaf->GetMinSize()){

    CoalesceOrRedistribute(deleaf,transaction);
  }
  buffer_pool_manager_->UnpinPage(deleaf->GetPageId(),true);
}
```

难点在于，CoalesceOrRedistribute

主要实现3项任务

1. ROOT PAGE，要做ADJUST ROOT。
2. 如果不是，首要要找到这个PAGE的兄弟PAGE。
3. 然后按照兄弟PAGE和当前PAGE的位置关系，来做2件事情。第一是当可以合并的时候，调用Coalesce。 如果不能合并，就意味着可以借节点。调用Redistribute

之后根据伪码依次实现其中的函数。

当时遇到的难点是，在递归调用CoalesceOrRedistribute函数时的条件判断，

```
if((*parent)->GetSize()<=(*parent)->GetMinSize())
  return  CoalesceOrRedistribute(*parent,transaction);
return false;
```

这里是parent->GetSize() <= parent->GetMinSize()

因为parent一定是内部节点，而内部节点第一个key是INVALID KEY,也就是说，当它的size等于最小size：2时，实际上只剩下一个key了，所以应当递归调用CoalesceOrRedistribute。

之后就是去leaf page和internal page中实现相关的move函数，按照书中的介绍来实现即可。

4.4 CATALOG MANAGER

Catalog Manager 负责管理和维护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。

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。

目录元信息

数据库中定义的表和索引在内存中以`TableInfo`和`IndexInfo`的形式表现，其维护了与之对应的表或索引的元信息和操作对象。以`IndexInfo`为例，它包含了这个索引定义时的元信息`meta_data_`，该索引对应的表信息`table_info_`，该索引的模式信息`key_schema_`和索引操作对象`index_`。除元信息`meta_data_`外，其它的信息（如`key_schema_`、`table_info_`等）都是通过反序列化后的元信息生成的。也就是说，为了能够将所有表和索引的定义信息持久化到数据库文件并在重启时从数据库文件中恢复，我们需要为表和索引的元信息`TableMetadata`和`IndexMetadata`实现序列化和反序列化操作。它们与`TableInfo`和`IndexInfo`定义在相同文件中。在序列化时，为了简便处理，我们为每一个表和索引都分配一个单独的数据页用于存储序列化数据。因此，在这样的设计下，我们同样需要一个数据页和数据对象`CatalogMeta`来记录和管理这些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。

实现函数：

- `CatalogMeta::SerializeTo(*buf)`
  - 与`RECORD MANAGER`中的类似，不再赘述
- `CatalogMeta::GetSerializedSize()`
  - 返回序列化所需字节数
- `CatalogMeta::DeserializeFrom(*buf, *heap)`

    - 不再赘述
- `IndexMetadata::SerializeTo(*buf)`
    - 不再赘述
- `IndexMetadata::GetSerializedSize()`
    - 返回序列化所需字节数
- `IndexMetadata::DeserializeFrom(*buf, *&index_meta, *heap)`
    - 不再赘叙
- `TableMetadata::SerializeTo(*buf)`
    - 不再赘叙
- `TableMetadata::GetSerializedSize()`
    - 返回序列化所需字节数
- `TableMetadata::DeserializeFrom(*buf, *&table_meta, *heap)`
    - 不再赘叙
- `IndexInfo::Init(*index_meta_data, *table_info, *buffer_pool_manager)`:
    - 传入事先创建好的`IndexMetadata`和从`CatalogManager`中获取到的`TableInfo`，创建索引本身的`key_schema_`和`index_`对象。`key_schema_`可以通过`Schema::ShallowCopySchema`来创建，`index_`通过获取的信息调用`IndexInfo::CreateIndex`创建。
- `IndexInfo::CreateIndex`:
    - 根据序列化索引模式时所需的最大字节数确定B+树索引的Keysize，调用B+树索引的构造函数构造即可，其空间来自`heap_`，便于管理。

表和索引的管理

在实现目录、表和索引元信息的持久化后，实现整个`CatalogManager`类。`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

实现函数如下：

- `CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager, LogManager *log_manager, bool init)`
    - 初次创建数据库实例时，只需要用`heap_`构建新实例即可。在后续重新打开数据库实例时，需要将catalogmetadata反序列化，并利用元信息先调用`LoadTable()`将所有表反序列化，再调用`LoadIndex()`将所有索引反序列化。
- `CatalogManager::~CatalogManager()`
    - 析构时调用`FlushCatalogMetaPage()`，将元数据序列化到磁盘中。并`delete heap_`释放空间
- `CatalogManager::CreateTable(const string &table_name, TableSchema *schema, Transaction *txn, TableInfo *&table_info)`
    - 先查看表是否已存在，然后利用信息构建`TableInfo`并初始化。为新表申请存元数据的页，序列化表的元数据。将信息更新到catalog和catalogmetadata中。
- `CatalogManager::GetTable(const string &table_name, TableInfo *&table_info)`
    - 利用表名和catalog信息获取表
- `CatalogManager::GetTable(const table_id_t table_id, TableInfo *&table_info)`
    - 利用`table_id`和catalog信息获取表

- `CatalogManager::GetTables(vector<TableInfo *> &tables)`
  - 获取所有表
- `CatalogManager::CreateIndex(const std::string &table_name, const string &index_name, const std::vector<std::string> &index_keys, Transaction *txn, IndexInfo *&index_info)`
  - 利用表名，索引名，索引键值映射创建索引。先检查表是否存在，再检查索引是不是早就存在了。然后判断`index_key`中的列是否都存在于表中，同时构建`key_map`。跟据已有信息构建`IndexInfo`，并初始化，为新索引申请存元数据的页，将索引元数据序列化。将信息更新到catalog和catalogmetadata中。
- `CatalogManager::GetIndex(const std::string &table_name, const std::string &index_name,IndexInfo *&index_info)`
  - 根据表名和索引名获取索引。先检查表是否存在，再检查索引是否存在。然后利用元数据找到该索引的`IndexInfo`
- `CatalogManager::GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes)`
  - 根据表名，利用catalog中的元数据找到该表上的所有索引
- `CatalogManager::DropTable(const string &table_name)`
  - 根据表名删除表，先检查表是否存在。找到表的`TableInfo`后释放堆表中的数据页，然后先删除表上的索引及对应的元数据，再删除表和对应的元数据。
- `CatalogManager::DropIndex(const string &table_name, const string &index_name)`
  - 根据表名和索引名删除索引，先检查表和索引是否存在，然后清除catalog中的信息以及metadata中的信息，并释放索引占用的页。
- `CatalogManager::FlushCatalogMetaPage()`
  - 将元信息序列化到磁盘中
- `CatalogManager::LoadTable(const table_id_t table_id, const page_id_t page_id)`
  - 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信息，构建`TableInfo`信息置于内存中。反序列化出信息后初始化`TableInfo`对象，并将信息更新到catalog中
- `CatalogManager::LoadIndex(const index_id_t index_id, const page_id_t page_id)`
  - 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索引信息，构建`IndexInfo`信息置于内存中。反序列化出信息后根据索引对应的`table_id`或许表的`TableInfo`，只有**当表存在时才允许构建索引。**得到所需信息后初始化`IndexInfo`对象，并将信息更新到catalog中

4.5 SQL EXECUTOR

功能分析

1. **语法树**

语法树最高层用于判断操作类型；向下一层用于分析各种条件，例如标记使用数据库（表）的名字的节点，标记需要显示属性名的节点，标记判断条件；再向下一层为显示属性（全部的话在上一层即可判断）和条件（连接符和比较操作符）的具体值。整体按语序进行排列，我们按照形状遍历即可。

2. **构造函数与退出**

执行器其实就是多个数据库引擎的一个`map`容器，所以其构造就是再次调用数据库的构造函数，重点在于修改退出机制，使其在退出时将所有数据库的名称存储于一个文件中，然后就可以在构造中利用名字重新构造数据库，注意重新构造时数据库`init`参数为`false`，因为不能清空原数据。

3. **显示功能与使用数据库**

包括数据库、表、索引在内的所有展示功能都是遍历存容器并去除成员的名字。数据库即遍历执行器内`map`容器，表需要找到当前使用的数据库再遍历，索引由于语法树无法给参数，本人采用遍历并输出数据库所有表对应索引的形式进行输出。

执行器本身有标记当前数据库的字符串变量，使用数据库即遍历所有数据库名，如果有匹配的即数据库存在，将当前数据库字符串修改即可。另外，其余函数在最开始都有判断数据库字符串是否为空（是否有`use database`），防止非法访问出现内存错误，包括各种存在的判断，下面的介绍都略过不谈。

4. **创建与删除**

主要分为数据库、表、索引的创建与删除。

数据库的创建与删除已经有给定的构造与析构函数，只要给定名字即可，注意创建新的数据库与从已有文件恢复数据库引擎的`init`参数是不同的即可。`true`时会导致删除本有的数据库文件建立新的空数据库，`false`会在原有文件的基础上重新打开数据库。

表的创建相对复杂点。得到表的名字后，需要遍历类型节点及其所有分支得到属性的名字和类型，通过属性名与类型先构造`Column`变量存入`vector`容器，再通过该容器构造一个`Schema`变量，声明一个空的表信息指针（`TableInfo*`），即可调用下层的函数进行表的建立。最后有的表声明了主键，而且由于语序排布，主键的声明都在最后，我们需要先拿到所有的声明，还要根据主键来给一些表的列`unique`的属性才能开始属性的构建（因为沿用现成的函数，很少进行增加和修改，部分类的成员一旦构造无法修改），包括得到属性后还需要创建主键的索引，该操作在下面索引部分详细介绍步骤。表的删除只需要获取表的名称即可调用下层的函数进行删除了。

索引建立是表的简化版，拿到表的名字和索引名字，再拿到建立索引的列名，判断`unique`后构建索引的`Schema`(和表的`Schema`是同一类)，调用底层函数即可实现建立。删除同样是通过名字调用底层函数即可，只是由于没有表的名字，需要遍历数据库的所有表。

5. **插入**

插入主要是`Field`以及`Row`的数据获取和构造，调用下层函数即可将`Row`插入对应表中，至于写入硬盘是底层维护的事。另外，我们需要针对`primary key`以及`unique`进行查重，具体方法为通过构造值进行索引的查找，查到了即插入失败，不能有重复值，未查到即可进行插入。最后插入表成功后还需要将对应值插入索引进行维护。

6. **选择、删除、更新**

这三种操作的主要部分都是条件的获取与判断，当然，选择额外加入了使用索引的判断与搜索操作。

选择需要先获取显示的列，后面对表的`Schema`进行遍历对应输出。更新需要获取更新的属性和值，用于后面调用函数。

而对于条件的使用，我们构造了一个新类`cmpData`包括了`Field`类以及一个`string`成员存储对应的比较操作符还有一个用于标记对应值需要比较的列在表中所处位置的"索引"（此索引非上面的索引）成员，而且实现了额外函数对于一组条件三个节点（操作符、属性名、属性值）进行构造变量。另外对于连接符导致的多种条件，我们使用一个`vector`容器来存储一组条件（即使用`and`进行连接，需要同时满足），再使用`vector<vector>`来存储多组（即使用`or`连接）条件，遍历时满足一组条件即可。由于语法树按语序进行构造，多组条件的实现比较麻烦，目前只实现了单个条件或连接符的操作。

具体查找过程，删除和更新都是通过迭代器对表进行遍历一一比较。而查找除此之外还有索引的使用，而由于框架的限制，目前只支持单个条件的相等查找，经过验证可以看出索引的查找效率确实远高于遍历。而对于索引的查找需要构造索引包含的属性对应`Row`然后调用函数进行查找，如果传入的`vectror`中存了有效的`RowId`说明查到了，接下来再利用`RowId`对表进行查找即可大大提升速度。

7. **执行文件**

通过对主函数的过程进行复制，我们便实现了文件执行，只要使用C++的`stream`便可较好复刻过程。只是注意判断文件的结束。

五、测试结果

MiniSQL中的测试文件测试结果如下，通过了框架提供的和我们自己添加的所有测试。对于SQL语句的测试在验收时已进行，步骤较多，此处就不再展示。

```
guokeyumao@LAPTOP-G8SE1MMQ  /mnt/d/Linux/MiniSql/build  part12345  ./test/minisql_test
[==========] Running 15 tests from 8 test suites.
[----------] Global test environment set-up.
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (5 ms)
[----------] 1 test from BufferPoolManagerTest (5 ms total)

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

[----------] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (13 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (6 ms)
[----------] 3 tests from CatalogTest (20 ms total)

[----------] 4 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (2 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (5 ms)
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (24 ms)
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (38 ms)
[----------] 4 tests from BPlusTreeTests (71 ms total)

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

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

```
[----------] 2 tests from DiskManagerTest
[ RUN      ] DiskManagerTest.BitMapPageTest
[       OK ] DiskManagerTest.BitMapPageTest (2 ms)
[ RUN      ] DiskManagerTest.FreePageAllocationTest
[       OK ] DiskManagerTest.FreePageAllocationTest (2105 ms)
[----------] 2 tests from DiskManagerTest (2108 ms total)

[----------] 1 test from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (229 ms)
[----------] 1 test from TableHeapTest (229 ms total)

[----------] Global test environment tear-down
[==========] 15 tests from 8 test suites ran. (2435 ms total)
[  PASSED  ] 15 tests.
```

六、总结与感悟

叶瑶波：

> 之前就听过几节CMU的课，也知道大名鼎鼎的`bustub`，所以对魔改版充满好奇。本次设计中我负责了三个模块，构建起了除索引外的整个底层，并在第五部分中写了更新和删除元组的函数，因此对底层与顶层都获得了较好的理解，在实现过程中对数据库的各种技术有了更直观的感受，只是很遗憾在课业压力下没能腾出时间探索更先进的技术，实现更好的优化。
>
> 从底层到顶层，一开始只是窥见一角，写了页的管理，后来写到`CATALOG MANAGER`时对整个数据库有了自顶向下的认知，对之前写的底层模块的理解也更加深入(于是发现了之前埋藏的几个小bug)。保持数据的持久性时进行的序列化和反序列操作非常新颖且有用，管理数据时元数据的使用，页面数据区的划分，使得我们对数据存储有了更深入的认知。层与层之间的无感知交互和每一层的良好封装都是设计应用时的重要思想。由于本学期刚解除C++，所以在编写代码的过程中遇到了很多语言方面的困难，也出了不少BUG，不过经过自主学习以及阅读框架中的代码，让我对C++的使用也更加娴熟了。
>
> 浙大的整个低年级计算机课程设计中缺乏对许多提高效率的工具的教授和实验文档的编写，在这点上MIT和伯克利等高校做的就很好。MiniSQL是我遇到的第一个有详细文档和各种环境搭建、工具、终端命令介绍的工程，非常赞。另外学git推荐`progit`，学各种工具推荐MIT的`missing semester`。
>
> 总的来说本次设计内容十分充实，也让我学到了很多新颖的思想。个人感觉目前框架聚集性有点高，底层和顶层分离的还不够清晰，会增加顶层设计者的负担，未来或许可以分离的更清晰些。或者在顶层模块的要求中稍微添加一些对底层的描述，或许有利于分工(当然小组成员积极交流更重要)。希望MiniSQL设计的越来越好:)

颜晗：

> `minisql`的实现就此结束，从一开始无从下手到最终实现了基本功能，感觉自己还是收获了很多，虽然没有亲自实现底层，但是由于执行器的实现离不开底层各种类与函数，不得不将整个框架与代码看了几遍。另外，`minisql`的复杂对于代码能力的提升也非常明显，对C++的各种功能特性的使用能力都得到了锻炼。
>
> 当然目前整个框架仍旧存在一些问题，包括部分提供的代码注释不足，容易导致学生理解偏差进而实现与理想有差距导致出现难以理解的错误，另外，代码中也有一些错漏？比如比较让人难以理解的为什么`TypeChar`类型有`GetData()`函数而另外两个没有实现，而且作为继承却能调用基类的……这种框架本身的代码一般都是通过`.h`文件看功能调用，一旦出现问题比较难查出。总之还是希望未来框架的代码和注释文档能够逐渐完善，减少学生无意义`debug`的过程，毕竟数据库本身也不是主要锻炼代码能力的课程。

熊儒海：

> 在本次实验中，我加深了对B+树操作的理解，也培养了编程能力，同时，也养成了良好的代码风格