

数据库系统

Mini SQL实验报告

2022.06.12

姓名	学号	分工
万士欣	3200106058	#1.2、#1.3、#2、#5
曹聿恺	3200103592	#1.4、#3
贾子钊	3200103592	#1.5、#4

MiniSql项目报告

1 项目概述

1.1 项目名称

小型关系数据库管理系统 (MiniSQL) 的设计与实现

1.2 项目需求

- 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
- 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。
- 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。
- 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。
- 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。

1.3 编写目的

- 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。
- 深入了解模式定义、查询处理、储存管理、索引管理、目录管理等相关实现技术
- 通过实验加深对DBMS及其内部实现技术的理解，实践系统软件开发的工程化方法。

1.3 项目背景

ZJUCSDB-MiniSQL框架，CMU15-445实验框架。

1.4 项目总体框架


```

1 class BitmapPage {
2     ...
3     static constexpr size_t MAX_CHARS = PageSize - 2 *
        sizeof(uint32_t);
4
5 private:
6     /** The space occupied by all members of the class should be
        equal to the PageSize */
7     [[maybe_unused]] uint32_t page_allocated_;
8     [[maybe_unused]] uint32_t next_free_page_;
9     [[maybe_unused]] unsigned char bytes[MAX_CHARS];
10 }

```

Bitmap Page有如下功能（函数）：

- **bool 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），具体步骤是：
 1. 根据 **page_allocated_** 判断是否有空闲页，没有则返回 **false**；
 2. 根据 **next_free_page_** 定位到bitmap中下一个可分配的空闲页的对应位，并置为1，如下：


```
1 bytes[next_free_page_ / 8] |= ( 1 << (next_free_page_ % 8) );
```
 3. **page_allocated_** 自增1，若此时 **page_allocated_** 已满则直接返回，反之则从旧的 **next_free_page_** 开始遍历寻找下一个空闲页并更新 **next_free_page_**。
- **BitmapPage::DeAllocatePage(page_offset)**：回收已经被分配的页，具体步骤是：
 1. 更新要释放页对应bit位为0，若已经为0，代表该页已经被释放，则返回 **false**；
 2. **page_allocated_--**；
 3. 若释放之前整个bitmap管理的所有页全满，则需要恢复 **next_free_page_** 为有效值，即刚刚释放的页对应位。
- **BitmapPage::IsPageFree(page_offset)**：通过bitmap判断给定的页是否是空闲（未分配）的。

2.1.2 Disk Manager

通过一个位图页加上一段连续的数据页（数据页的数量取决于位图页最大能够支持的比特数）从而可实现对磁盘文件（DBFile）中数据页进行分配和回收。

进一步地，为了扩充一个文件可以存储的数据（在上述描述中，假定数据页的大小为4KB，一个位图页中的每个字节都用于记录，那么这个位图页最多能够管理32768个数据页，也就是说，这个文件最多只能存储 $4K * 8 * 4KB = 128MB$ 的数据），我们将一个位图页加一段连续的数据页看成数据库文件中的一个分区（Extent），再通过一个额外的元信息页来记录这些分区的信息，如下：

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

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，包括分区的数量、每个分区中已经分配的页的数量等，值得注意的是，DiskFileMetaPage 类中的最后的 extent_used_page_ 为柔性数组，具体可参考C语言柔性数组 - wuyudong - 博客园 (cnblogs.com)，如下：

```
1 class DiskFileMetaPage {
2     ...
3 public:
4     uint32_t num_allocated_pages_{0};
5     uint32_t num_extents_{0}; // each extent consists with a bit
    map and BIT_MAP_SIZE pages
6     uint32_t extent_used_page_[0];
7 }
```

在具体实现中被存储在 DiskManager 类的 char meta_data_[PAGE_SIZE] 中，要使用（修改和读取）时，通过 reinterpret_cast 将 char 数组指针类型强制转化为 DiskFileMetaPage 类指针从而调用，如下：

```
1 auto * metaPage = reinterpret_cast<DiskFileMetaPage *>
    (meta_data_);
```

另外，由于元数据所占用的数据页实际上是不存储数据库数据的，而它们实际上又占据了数据库文件中的数据页，换言之，实际上真正存储数据的数据页是不连续的。举一个简单例子，假设每个分区能够支持3个数据页，那么实际上真正存储数据的页只有：2, 3, 4, 6, 7, 8..., 如下：

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	

但实际上，对于上层的Buffer Pool Manager来说，希望连续分配得到的页号是连续的 (0, 1, 2, 3...)，为此，在Disk Manager中，需要对页号做一个映射（映射成上表中的逻辑页号），这样才能使得上层的Buffer Pool Manager对于Disk Manager中的页分配是无感知的。

综上，对于此模块，须实现以下功能（函数）：

- **DiskManager::AllocatePage()**：从磁盘中分配一个空闲页，并返回空闲页的逻辑页号，具体步骤如下：
 1. 根据 metaPage->num_allocated_pages_ 判断是否有空闲页；

2. 若当前所有分区已满，则创建新的分区：

```
1 if (metaPage->num_allocated_pages_ == metaPage->num_extents_ *  
    BITMAP_SIZE) metaPage->num_extents_++;
```

3. `num_allocated_pages_` 自增；

4. 遍历找到第一个有空闲页的分区，更新 `metaPage`，将对应分区的位图页从磁盘中读出并更新后再写回磁盘，将 `meta_data_` 也写回磁盘。

- `DiskManager::DeAllocatePage(logical_page_id)`：释放磁盘中逻辑页号对应的物理页，具体步骤如下：

1. 根据 `logical_page_id` 得出其所处分区，从磁盘读出分区的位图页释放对应页（见2.1.1），并再写回磁盘；

2. 更新 `meta_data_`。

- `DiskManager::IsPageFree(logical_page_id)`：判断该逻辑页号对应的数据页是否空闲，同样是根据 `logical_page_id` 得出其所处分区，从磁盘读出分区的位图页判断是否空闲。

- `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，具体可根据上面表中的规律实现实现，如下：

```
1 page_id_t DiskManager::MapPageId(page_id_t logical_page_id) {  
2     return 2 + logical_page_id / 3 + logical_page_id;  
3 }
```

2.1.3 LRU REPLACER

`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。在本节中，我们需要实现一个基于LRU替换算法的 `LRUReplacer`，它在 `src/include/buffer/lru_replacer.h` 中被定义，其扩展了抽象类 `Replacer`（在 `src/include/buffer/replacer.h` 中被定义）。 `LRUReplacer` 的大小默认与Buffer Pool的大小相同。

`LRUReplacer` 类中的私有成员变量如下所示：

```
1 class LRUReplacer : public Replacer {  
2 private:  
3     size_t len = 0;  
4     std::list<frame_id_t> lru_list;  
5     std::unordered_map<frame_id_t,  
6         std::list<frame_id_t>::iterator> lru_map;  
7 };
```

除了上述私有变量外，它还提供了一些公开方法：

- `LRUReplacer::Victim(frame_id_t *frame_id)` 主要负责替换与所有被跟踪的页相比最近最少被访问的页，将其页帧号存储在输出参数 `frame_id` 中。该函数的输出

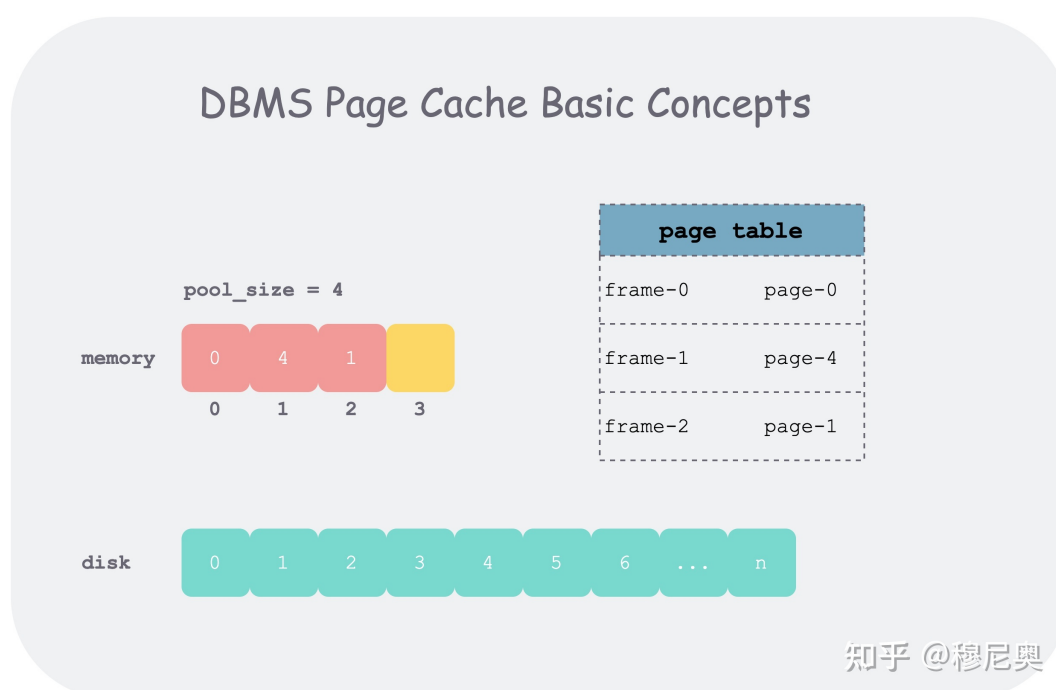
主要负责表明是否有可替换的页，如果有则返回true否则返回false。

- `LRUReplacer::Pin(frame_id)` 用于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。
- `LRUReplacer::Unpin(frame_id)` 用于将数据页解除固定，放入 `lru_list_` 中，使之可以在必要时被 `Replacer` 替换掉。
- `LRUReplacer::Size()` 用于返回当前 `LRUReplacer` 中能够被替换的数据页的数量。

2.1.4 Buffer Pool Manager

2.1.4.1 Buffer Pool介绍

Buffer Pool 是内存空间中的一段区域，用来储存从Disk Manager中获取的数据页。一个Buffer Pool有若干的“槽”（frame），用来储存磁盘中对应的一段存储空间（page）。



当这一段page被从磁盘中取出放置到buffer pool的槽中的时候，会以 `Page` 类的形式，为其附加一些元信息：`pin_count_`，`is_dirty_`，`page_id_`。

```
1 private:
2     /** Zeroes out the data that is held within the page. */
3     inline void ResetMemory() { memset(data_, OFFSET_PAGE_START,
4         PAGE_SIZE); }
5
6     /** The actual data that is stored within a page. */
7     char data_[PAGE_SIZE]{};
8     /** The ID of this page. */
9     page_id_t page_id_ = INVALID_PAGE_ID;
10    /** The pin count of this page. */
11    int pin_count_ = 0;
12    /** True if the page is dirty, i.e. it is different from its
13        corresponding page on disk. */
14    bool is_dirty_ = false;
```

```
13  /** Page latch. */
14  ReaderWriterLatch rwlatch_;
```

2.1.4.2 Buffer Pool Manager概述

Buffer Pool Manager 负责上文介绍的Buffer Pool的管理，主要功能包括：

1. 根据需求，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。
5. 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)，在本实验中，数据页的大小默认为 4KB。

Buffer Pool Manager也由对应的类来实现

```
1  private:
2      size_t pool_size_;                // number of pages in
    buffer pool
3      Page *pages_;                    // array of pages
4      DiskManager *disk_manager_;      // pointer to the disk
    manager.
5      std::unordered_map<page_id_t, frame_id_t> page_table_; // to
    keep track of pages
6      Replacer *replacer_;             // to find an unpinned
    page for replacement
7      std::list<frame_id_t> free_list_; // to find a free page
    for replacement
8      recursive_mutex latch_;
```

- `size_t pool_size_` 是buffer pool 的规模，也就是其中槽的个数，在本实验中默认值为1024
- `std::unordered_map<page_id_t, frame_id_t> page_table_` 是记录槽的编号 (frame id) 和储存的页的编号 (page id) 的对应关系的。
- `Page * pages_` 是槽中以Page类形式储存在内存中的磁盘页的数组。
- `Replacer *replacer_` 是封装好LRU替换策略的模块
- `std::list<frame_id_t> free_list_` 是当前空闲的槽的编号的集合，用于寻找空槽。

2.1.4.3 Buffer Pool Manager具体实现

- `AllocatePage()` 调用disk manager从磁盘中分配一个页
- `DeallocatePage(page_id_t page_id)` 调用disk manager将指定的页在磁盘中释放
- `FetchPage(page_id_t page_id)` 抓取磁盘中指定编号的页放到内存槽里

- 如果内存槽没有满，则直接将此页放入空余内存槽中
- 如果内存槽已经满了，则使用LRU策略替换出一个页。需要注意，pin count不为0的页不能被替换出去，如果所有的槽都无法被替换，则返回false
- 在映射表中记录page id和frame id的对应关系，初始化Page类中其他的属性
- **NewPage(page_id_t &page_id)**
 - 按照上述相同的策略寻找槽中空位
 - 调用 **AllocatePage** 从磁盘中分配新的页
 - 在映射表中记录page id和frame id的对应关系，初始化Page类中其他的属性
 - 返回页编号
- **DeletePage(page_id_t page_id)**
 - 从映射关系同检查是否存在这个页，如果不存在返回false。如果存在找到所存的槽编号
 - 如果pin count不为0，说明还在被其他进程使用，无法删除
 - 如果is dirty标记存在，将这个页写回磁盘
 - 调用 **DeallocatePage(page_id_t page_id)** 从磁盘中释放。
 - 清空槽，标记该槽编号为空
- **UnpinPage(page_id_t page_id, bool is_dirty)**
 - 将槽中page id对应的pin count -1
 - 如果该页不存在在槽里，或者pin count不大于0，返回false
- **FlushPage(page_id_t page_id)** 将page里面的数据写回磁盘

2.2 RECORD MANAGER

在MiniSQL的设计中，Record Manager负责管理数据表中所有的记录，它能够支持记录的插入、删除与查找操作，并对外提供相应的接口。

2.2.1 相关概念介绍

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，如下：

```

1  class Column {
2      ...
3  private:
4      std::string name_;
5      TypeId type_;
6      uint32_t len_{0};          // for char type this is the maximum
                                // byte length of the string data,
7      // otherwise is the fixed size
8      uint32_t table_ind_{0};    // column position in table
9      bool nullable_{false};    // whether the column can be null
10     bool unique_{false};       // whether the column is unique
11 };

```

- 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, 如下:

```
1 class Schema {
2     ...
3     private:
4         std::vector<Column *> columns_;    /** don't need to delete
        pointer to column */
5 };
6
7 using IndexSchema = Schema;
8 using TableSchema = Schema;
```

- 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, 需要注意的是**枚举**的使用, 如下:

```
1 enum TypeId {
2     kTypeInvalid = 0,
3     kTypeInt,
4     kTypeFloat,
5     kTypeChar,
6     KMaxTypeId = kTypeChar
7 };
8
9 class Field {
10     friend class Type;
11     friend class TypeInt;
12     friend class TypeChar;
13     friend class TypeFloat;
14     ...
15 protected:
16     union Val {
17         int32_t integer_;
18         float float_;
19         char *chars_;
20     } value_;
21     TypeId type_id_;
22     uint32_t len_;
23     bool is_null_{false};
24     bool manage_data_{false};
25 };
```

- 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, 如下:

```

1 class Row {
2     ...
3 private:
4     RowId rid_{};
5     std::vector<Field *> fields_;    /** Make sure that all fields are
        created by mem heap */
6     MemHeap *heap_{nullptr};
7 };

```

2.2.2 Serialize and Deserialize

为了能够持久化存储上面提到的 `Row`、`Field`、`Schema` 和 `Column` 对象，我们需要提供一种能够将这些对象序列化成为字节流（`char*`）的方法，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，我们同样需要能够提供一种反序列化的方法，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。

为了确保我们的数据能够正确存储，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。

在本节中，需要完善 `Row`、`Schema` 和 `Column` 对象各自的 `SerializeTo`、`DeserializeFrom` 和 `GetSerializedSize` 方法，具体有：

- `uint32_t Row::SerializeTo(*buf, *schema)`
 1. 序列化 `MAGIC_NUM`；
 2. 通过遍历 `vector<Field *> fields_` 得到 `field_num` 和 `null_bitmap`，后者标记了为 `null` 的 `Field`，注意不用序列化 `rowid`；
 3. 再次遍历调用 `Field` 的析构函数依次进行析构；
 4. 最后返回过程中 `buf` 指针向前推进了多少个字节（可通过调用函数 `GetSerializedSize` 得到）。
- `uint32_t Row::DeserializeFrom(*buf, *schema)`，反向实现序列化的过程即可。
- `uint32_t Row::GetSerializedSize(*schema)`，如下：

```

1 uint32_t Row::GetSerializedSize(Schema *schema) const {
2     uint32_t buf_add = 2 * sizeof(int64_t);
3     int64_t field_num = fields_.size();
4     for (int64_t i = 0; i < field_num; i++)
5     {
6         buf_add += fields_[i]→GetSerializedSize();
7     }
8     return buf_add;
9 }

```

- `uint32_t Column::SerializeTo(*buf)`
 1. 序列化 `MAGIC_NUM`；

2. 先序列化列的 `name` 的长度，再将 `string` 类的 `name_` 直接变成 `char *` 字符串从而序列化；
 3. 继续依次序列化列的类型 `type_`、长度 `len_`、在table中的序号 `table_ind_` 以及是否为空 `nullable_` 和是否独一 `nullable_`；
 4. 最后返回过程中 `buf` 指针向前推进了多少个字节。
- `uint32_t Column::DeserializeFrom(*buf, *&column, *heap)`，反向实现序列化的过程即可。
 - `uint32_t Column::GetSerializedSize()`，如下：

```
1 uint32_t Column::GetSerializedSize() const {
2     return static_cast<uint32_t>( 4 * sizeof(uint32_t) +
    name_.length() + sizeof(int) + 2 * sizeof(bool) );
3 }
```

- `uint32_t Schema::SerializeTo(*buf)`
 1. 序列化 `MAGIC_NUM`；
 2. 序列化 `column` 的数量；
 3. 遍历 `vector<Column *> columns_` 依次序列化每个 `schema`；
 4. 最后返回过程中 `buf` 指针向前推进了多少个字节。
- `uint32_t Schema::DeserializeFrom(*buf, *&schema, *heap)`，反向实现序列化的过程即可。
- `uint32_t Schema::GetSerializedSize()`，如下：

```
1 uint32_t Schema::GetSerializedSize() const {
2     return static_cast<uint32_t>( sizeof(uint32_t) + sizeof(int) +
    columns_.size() * columns_[0]→GetSerializedSize() );
3 }
```

另外，对于 `Row` 类型对象的序列化，在反序列化每一个 `Field` 时，需要将自身的 `heap_` 作为参数传入到 `Field` 类型的 `Deserialize` 函数中，这也意味着所有反序列化出来的 `Field` 的内存都由该 `Row` 对象维护。对于 `Column` 和 `Schema` 类型对象的反序列化，将使用 `MemHeap` 类型的对象 `heap` 来分配空间，分配后新生成的对象于参数 `column` 和 `schema` 中返回。

2.2.3 Table Heap

2.2.3.1 RowId

对于数据表中的每一行记录，都有一个唯一标识符 `RowId`（`src/include/common/rowid.h`）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**），可以快速定位到其对应的记录位于物理文件的哪个位置。

具体如下：

```
1 class RowId {
2     ...
3 private:
4     page_id_t page_id_{INVALID_PAGE_ID};
5     uint32_t slot_num_{0}; // logical offset of the record in
    page, starts from 0. eg:0, 1, 2...
6 };
7
8 static const RowId INVALID_ROWID = RowId(INVALID_PAGE_ID, 0);
```

2.2.3.2 Table Page and Heap

TableHeap，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。在 **TableHeap** 中只记录了双向链表中的 **first_page_id_**，可依此遍历整个双向链表。同时还记录的对应表的 **schema**，即一个 **TableHeap** 对应一个 table。
buffer_pool_manager_ 用于 **TablePage** 在缓冲区的管理，包括将 **TablePage** 的创建、从磁盘中抓取和写回磁盘等，具体如下：

```
1 class TableHeap {
2     ...
3 private:
4     BufferPoolManager *buffer_pool_manager_;
5     page_id_t first_page_id_;
6     Schema *schema_;
7     [[maybe_unused]] LogManager *log_manager_;
8     [[maybe_unused]] LockManager *lock_manager_;
9 };
```

堆表中的每个数据页 **TablePage** 是 **Page** 的继承类，**Page** 主要组成成为一个 **char data_[PAGE_SIZE]{}。**而每个 **TablePage** 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 (见下图)。

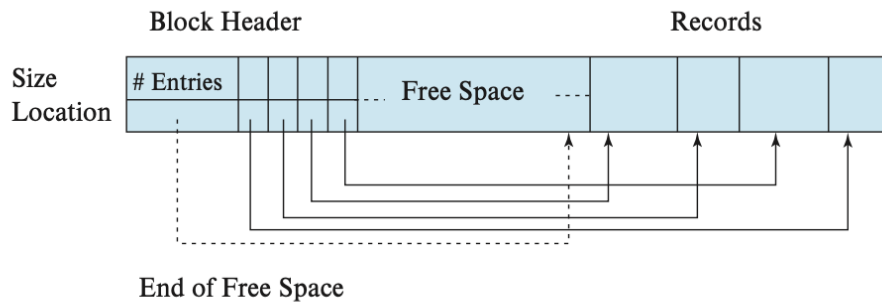


Figure 13.6 Slotted-page structure.

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，如下：

```

1  /**
2   * Header format (size in bytes):
3   * -----
4   * | PageId (4)| LSN (4)| PrevPageId (4)| NextPageId (4)|
   FreeSpacePointer(4) |
5   * -----
6   * -----
7   * | TupleCount (4) | Tuple_1 offset (4) | Tuple_1 size (4) |
   ... |
8   * -----
9   */
10 class TablePage : public Page {
11     ...
12 private:
13     static_assert(sizeof(page_id_t) == 4);
14     static constexpr uint64_t DELETE_MASK = (1U << (8 *
   sizeof(uint32_t) - 1));
15     static constexpr size_t SIZE_TABLE_PAGE_HEADER = 24;
16     static constexpr size_t SIZE_TUPLE = 8;
17     static constexpr size_t OFFSET_PREV_PAGE_ID = 8;
18     static constexpr size_t OFFSET_NEXT_PAGE_ID = 12;
19     static constexpr size_t OFFSET_FREE_SPACE = 16;
20     static constexpr size_t OFFSET_TUPLE_COUNT = 20;
21     static constexpr size_t OFFSET_TUPLE_OFFSET = 24;
22     static constexpr size_t OFFSET_TUPLE_SIZE = 28;
23
24 public:
25     static constexpr size_t SIZE_MAX_ROW = PAGE_SIZE -
   SIZE_TABLE_PAGE_HEADER - SIZE_TUPLE;
26 };

```

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。

此外，在堆表中还需要实现迭代器

`TableIterator` (`src/include/storage/table_iterator.h`)，以便上层模块遍历堆表中的所有记录，迭代器中包含了一个指向 `Row` 对象的指针，具体如下：

```
1 class TableIterator {
2 public:
3     ...
4     inline bool operator==(const TableIterator &itr) const {
5         return this->row_>GetRowId() == itr.row_>GetRowId();
6     }
7
8     inline bool operator!=(const TableIterator &itr) const {
9         return !(this->row_>GetRowId() == itr.row_>GetRowId());
10    }
11    const Row &operator*(); // 返回Row对象的引用
12
13    Row *operator->(); // 返回Row对象的指针
14
15    TableIterator &operator++(); // 更新指向下一个Row
16
17 private:
18     TableHeap *tableheap_;
19     Row *row_;
20 };
```

具体实现的功能（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_`），具体步骤如下：
 1. 首先遍历当前 `TableHeap` 中的双向链表，依此尝试插入记录，若插入成功则返回 `true`；
 2. 若当前双向链表中所有 `TablePage` 都无法容纳要插入的记录，则创建新的 `TablePage` 并插入记录，最后将新 `TablePage` 插入双向链表的尾部即可。
- `TableHeap:ApplyDelete(&rid, *txn)`：根据 `rid` 定位要删除的记录所在的 `TablePage` 以及在 `TablePage` 中的具体位置并删除即可。
- `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，具体步骤如下：
 1. 尝试直接在其所在的 `TablePage` 中更新，若更新成功，则返回 `true`；
 2. 若更新不成功，则直接删除要更新的旧记录，再插入新纪录即可。
- `TableHeap:GetTuple(*row, *txn)`：获取 `RowId` 为 `row->rid_` 的记录；
- `TableHeap:FreeHeap()`：销毁整个 `TableHeap` 并释放这些数据页；

- `TableHeap::Begin()` : 获取堆表的首迭代器, 具体通过从头遍历 `TableHeap` 找到第一个 `Row`, 并将根据该 `Row` 为迭代器的 `Row` 指针分配一个相同的 `Row` (注意此处不能将迭代器的 `Row` 指针直接指向 `TableHeap` 中的 `Row` 的原因是在 `TablePage` 中存储的 `Row` 其实是 `Row` 序列化后的char字符串)。
- `TableHeap::End()` : 获取堆表的尾迭代器, 直接为迭代器的 `Row` 指针分配一个 `INVALID_ROW` 即可;
- `TableIterator::operator++()` : 遍历 `TableHeap` 移动到下一条记录, 通过 `++iter` 调用。

2.3 INDEX MANAGER

2.3.1 BPlusTreePage

`BPlusTreePage` 是 `BPlusTreeInternalPage` 和 `BPlusTreeLeafPage` 类的公共父类, 它包含了中间结点和叶子结点共同需要的数据:

- `page_type_` : 标记数据页是中间结点还是叶子结点;
- `lsn_` : 数据页的日志序列号, 目前不会用到, 如果之后感兴趣做Crash Recovery相关的内容需要用到;
- `size_` : 当前结点中存储Key-Value键值对的数量;
- `max_size_` : 当前结点最多能够容纳Key-Value键值对的数量;
- `parent_page_id_` : 父结点对应数据页的 `page_id` ;
- `page_id_` : 当前结点对应数据页的 `page_id` 。

```
1 class BPlusTreePage {
2 private:
3     [[maybe_unused]] IndexPageType page_type_;
4     [[maybe_unused]] lsn_t lsn_;
5     [[maybe_unused]] int size_;
6     [[maybe_unused]] int max_size_;
7     [[maybe_unused]] page_id_t parent_page_id_;
8     [[maybe_unused]] page_id_t page_id_;
9 };
```

在 `BPlusTreePage.cpp` 中, 我们实现了几个公开方法:

- `BPlusTreePage::IsLeafPage()` 与 `BPlusTreePage::SetPageType(IndexPageType page_type)` 用于判断以及设置该页面为叶节点还是中间节点
- `BPlusTreePage::IsRootPage()` 用于判断该页面是否为根节点
- `BPlusTreePage::GetSize()` 与 `BPlusTreePage::SetSize(int size)` 分别用于获取和设置页面大小
- `BPlusTreePage::GetMaxSize()` 与 `BPlusTreePage::SetMaxSize(int max_size)` 分别用于获取和设置页面的最大大小
- `BPlusTreePage::IncreaseSize(int amount)` 用于增加页面大小
- `BPlusTreePage::GetMinSize()` 用于获取页面的最小大小

- `BPlusTreePage::GetParentPageId()` 与 `BPlusTreePage::SetParentPageId(page_id_t parent_page_id)` 分别用于获取和设置页面的父节点ID
- `BPlusTreePage::GetPageId()` 与 `BPlusTreePage::SetPageId(page_id_t page_id)` 分别用于获取和设置页面的该页面ID

2.3.2 BPlusTreeInternalPage

中间结点 `BPlusTreeInternalPage` 不存储实际的数据，它只按照顺序存储m个键和m+1个指针（这些指针记录的是子结点的 `page_id`）。由于键和指针的数量不相等，因此我们需要将第一个键设置为INVALID，也就是说，顺序查找时需要从第二个键开始查找。在任何时候，每个中间结点至少是半满的（Half Full）。当删除操作导致某个结点不满足半满的条件，需要通过合并（Merge）相邻两个结点或是从另一个结点中借用（移动）一个元素到该结点中（Redistribute）来使该结点满足半满的条件。当插入操作导致某个结点溢出时，需要将这个结点分裂成为两个结点。

在 `src/include/storage/page/b_plus_tree_internal_page.h` 和 `src/page/b_plus_tree_internal_page.cpp` 两个件中，`BPlusTreeInternalPage` 类被实现，它继承自 `BPlusTreePage` 类，包含了若干公开方法与一些私有属性。

```
1 class BPlusTreeInternalPage : public BPlusTreePage {
2 private:
3     void CopyNFrom(MappingType *items, int size, BufferPoolManager
      *buffer_pool_manager);
4     void CopyLastFrom(const MappingType &pair, BufferPoolManager
      *buffer_pool_manager);
5     void CopyFirstFrom(const MappingType &pair, BufferPoolManager
      *buffer_pool_manager);
6     MappingType array_[0];
7 };
```

在 `BPlusTreeInternalPage` 类中，主要实现的是一些中间节点操作相关的公开方法：

- `BPlusTreeInternalPage::Init(page_id_t page_id, page_id_t parent_id = INVALID_PAGE_ID, int max_size = INTERNAL_PAGE_SIZE)` 用于初始化中间节点，设置该节点的page_ID, parent_ID以及max_size属性。
- `BPlusTreeInternalPage::KeyAt(int index)` 用于查找 `array_[]` 数组中index相对应的key。
- `BPlusTreeInternalPage::SetKeyAt(int index, const KeyType &key)` 用于设置 `array_[]` 数组中index位置的key。
- `BPlusTreeInternalPage::ValueIndex(const ValueType &value)` 用于查找 `array_[]` 数组中value所在位置的index。直接通过循环语句对数组进行遍历即可
- `BPlusTreeInternalPage::ValueAt(int index)` 用于查找 `array_[]` 数组中index位置相对应的value。
- `BPlusTreeInternalPage::Lookup(const KeyType &key, const KeyComparator &comparator)` 用于查找 `array_[]` 数组中key所在位置的

value。需要注意的是，这里的查找不是精确查找，而是返回最后一个不大于传入key的 `array_[]` 中的value即可。

- `BPlusTreeInternalPage::PopulateNewRoot(const ValueType &old_value, const KeyType &new_key, const ValueType &new_value)` 用于填充新的根节点。将传入的new_key与new_value设置为该中间节点页面 `array_[]` 数组的1号位，再将0号位的value设置为old_value，并更改页面大小即可。
- `BPlusTreeInternalPage::InsertNodeAfter(const ValueType &old_value, const KeyType &new_key, const ValueType &new_value)` 用于在数组中插入新的键值对。找到value在数组中的位置，然后将它后面的节点都向后挪动，再在该位置后一位插入新的键值对即可。
- `BPlusTreeInternalPage::Remove(int index)` 用于移除数组中index位置的节点。
- `BPlusTreeInternalPage::RemoveAndReturnOnlyChild()` 用于将该中间节点移除并返回它唯一的儿子节点。只要将它的页面尺寸设置为0并返回数组中0号位中的value即可。
- `BPlusTreeInternalPage::MoveAllTo(BPlusTreeInternalPage *recipient, const KeyType &middle_key, BufferPoolManager *buffer_pool_manager)` 用于将该节点的所有键值对移动到目标节点中去。调用后文实现的键值对移动函数，并重新设置两个页面的大小即可。
- `BPlusTreeInternalPage::MoveHalfTo(BPlusTreeInternalPage *recipient, BufferPoolManager *buffer_pool_manager)` 用于将该节点的一半键值对移动到目标节点中去。调用后文实现的键值对移动函数，并重新设置两个页面的大小即可。
- `BPlusTreeInternalPage::MoveFirstToEndOf(BPlusTreeInternalPage *recipient, const KeyType &middle_key, BufferPoolManager *buffer_pool_manager)` 用于将该节点的第一个节点移动到目标节点的最后一个节点处即可。调用后文实现的 `CopyLastFrom()` 函数并重新设置页面大小即可。
- `BPlusTreeInternalPage::MoveLastToFrontOf(BPlusTreeInternalPage *recipient, const KeyType &middle_key, BufferPoolManager *buffer_pool_manager)` 用于将该节点的最后一个节点移动到目标节点的第一个节点处即可。调用后文实现的 `CopyFirstFrom()` 函数并重新设置页面大小即可。

除去上述公开方法外，该类还有三个私有方法，分别是 `CopyNFrom()`，`CopyLastFrom()`，`CopyFirstFrom()` 三个函数。它们主要负责操纵 `buffer_pool_manager` 对两个不同页面之间的键值对进行拷贝。

2.3.3 BPlusTreeLeafPage

叶结点 `BPlusTreeLeafPage` 存储实际的数据，它按照顺序存储个键和 m 个值，其中键由一个或多个 `Field` 序列化得到（参考#3.2.4），在 `BPlusTreeLeafPage` 类中用模板参数 `KeyType` 表示；值实际上存储的是 `RowId` 的值，它在 `BPlusTreeLeafPage` 类中用模板参数 `ValueType` 表示。叶结点和中间结点一样遵循着键值对数量的约束，同样也需要完成对应的合并、借用和分裂操作。

在 `src/include/storage/page/b_plus_tree_leaf_page.h` 和 `src/page/b_plus_tree_leaf_page.cpp` 两个件中，`BPlusTreeLeafPage` 类被实现，它继承自 `BPlusTreePage` 类，包含了若干公开方法与一些私有属性。

```

1 class BPlusTreeLeafPage : public BPlusTreePage {
2 private:
3     void CopyNFrom(MappingType *items, int size);
4     void CopyLastFrom(const MappingType &item);
5     void CopyFirstFrom(const MappingType &item);
6     page_id_t next_page_id_;
7     MappingType array[0];
8 };

```

在 `BPlusTreeLeafPage` 类中，主要实现的是一些叶节点操作相关的公开方法：

- `BPlusTreeLeafPage::Init(page_id_t page_id, page_id_t parent_id = INVALID_PAGE_ID, int max_size = LEAF_PAGE_SIZE)` 用于初始化叶节点，设置该节点的page_ID, parent_ID以及max_size属性。
- `BPlusTreeLeafPage::GetNextPageId()` 与 `BPlusTreeLeafPage::SetNextPageId(page_id_t next_page_id)` 用于获取/设置叶节点的下一个兄弟结点ID。
- `BPlusTreeLeafPage::KeyAt(int index)` 用于查找 `array_[]` 数组中index相对应的key。
- `BPlusTreeLeafPage::KeyIndex(const KeyType &key, const KeyComparator &comparator)` 用于查找数组中第一个不小于传入key的index。直接对数组进行遍历查找即可。
- `BPlusTreeLeafPage::GetItem(int index)` 用于查找 `array_[]` 数组中index位置相对应的键值对。
- `BPlusTreeLeafPage::Insert(const KeyType &key, const ValueType &value, const KeyComparator &comparator)` 用于在该叶节点中插入键值对。遍历叶节点中的数组，找到合适的位置进行插入即可。
- `BPlusTreeLeafPage::Lookup(const KeyType &key, ValueType &value, const KeyComparator &comparator)` 用于查找 `array_[]` 数组中key所在位置的value。需要注意的是，这里的查找是精确查找，结果的value放到传入的参数中。返回值为一个bool类型的变量用于判定是否找到。
- `BPlusTreeLeafPage::RemoveAndDeleteRecord(const KeyType &key, const KeyComparator &comparator)` 用于将key对应的键值对从叶节点的数组中删除。
- `BPlusTreeLeafPage::MoveAllTo(MoveAllTo(BPlusTreeLeafPage *recipient))` 用于将该节点的所有键值对移动到目标节点中去。调用后文实现的键值对移动函数，并重新设置两个页面的大小即可。
- `BPlusTreeLeafPage::MoveHalfTo(BPlusTreeLeafPage *recipient)` 用于将该节点的一半键值对移动到目标节点中去。调用后文实现的键值对移动函数，并重新设置两个页面的大小即可。
- `BPlusTreeLeafPage::MoveFirstToEndOf(BPlusTreeInternalPage *recipient)` 用于将该节点的第一个节点移动到目标节点的最后一个节点处即可。调用后文实现的 `CopyLastFrom()` 函数并重新设置页面大小即可。
- `BPlusTreeLeafPage::MoveLastToFrontOf(BPlusTreeInternalPage *recipient)` 用于将该节点的最后一个节点移动到目标节点的第一个节点处即可。调用后文实现的 `CopyFirstFrom()` 函数并重新设置页面大小即可。

除去上述公开方法外，该类还有三个私有方法，分别是 `CopyNFrom()` ，
`CopyLastFrom()` ， `CopyFirstFrom()` 三个函数。它们主要负责对两个不同
页面之间的键值对进行拷贝。

2.3.4 BPlusTree

在完成B+树结点的数据结构设计后，接下来需要完成B+树的创建、插入、删除、查找
和释放等操作。注意，所设计的B+树只能支持 `Unique Key` ，这也意味着，当尝试向
B+树插入一个重复的Key-Value键值对时，将不能执行插入操作并返回 `false` 状
态。当一些写操作导致B+树索引的根节点发生变化时，需要调用
`BPLUSTREE_TYPE::UpdateRootPageId` 完成 `root_page_id` 的变更和持久化。

在B+树的数据结构中，主要有如下私有变量以及函数成员。

```
1  class BPlusTree {
2  private:
3      void StartNewTree(const KeyType &key, const ValueType
        &value);
4      bool InsertIntoLeaf(const KeyType &key, const ValueType
        &value, Transaction *transaction = nullptr);
5      void InsertIntoParent(BPlusTreePage *old_node, const KeyType
        &key, BPlusTreePage *new_node, Transaction *transaction =
        nullptr);
6      template<typename N>
7          N *Split(N *node);
8      template<typename N>
9          bool CoalesceOrRedistribute(N *node, Transaction *transaction
        = nullptr);
10     template<typename N>
11         bool Coalesce(N **neighbor_node, N **node,
        BPlusTreeInternalPage<KeyType, page_id_t, KeyComparator>
        **parent, int index, Transaction *transaction = nullptr);
12     template<typename N>
13         void Redistribute(N *neighbor_node, N *node, int index);
14         bool AdjustRoot(BPlusTreePage *node);
15         void UpdateRootPageId(int insert_record = 0);
16         void ToGraph(BPlusTreePage *page, BufferPoolManager *bpm,
        std::ofstream &out) const;
17         void ToString(BPlusTreePage *page, BufferPoolManager *bpm)
        const;
18         index_id_t index_id_;
19         page_id_t root_page_id_;
20         BufferPoolManager *buffer_pool_manager_;
21         KeyComparator comparator_;
22         int leaf_max_size_;
23         int internal_max_size_;
24     };
```

`StartNewTree` 负责构建出一棵新的树，`InsertIntoLeaf` 与 `InsertIntoParent` 分别负责将键值对的内容插入叶节点与其对应的父节点中。`AdjustRoot` 负责在根节点中进行相应调整，`UpdateRootPageId` 则负责对根节点ID进行插入或更新。

`Split` 负责对其中键值对过多的节点进行分裂。`CoalesceOrRedistribute` 负责在移除节点的时候对树的结构进行改变，对节点进行合并或重构操作。而 `Coalesce` 与 `Redistribute` 则将合并以及重构的操作进行细化，调用中间节点与叶节点中的 `MoveAllTo` `MoveHalfTo` `MoveFirstToEndOf` `MoveLastToFrontOf` 几个函数对叶节点中储存的数据进行改变。由于这些都是模板函数，所以在使用时我们要区分不同的节点类型进行不同的操作。

而该类中的公开方法如下所示：

- `BPlusTree::BPlusTree()` 是B+树的构造函数，负责对B+树进行构建。如果bpm里存有根页面的数据，则应根据根页面对B+树进行相应的恢复操作。
- `BPlusTree::IsEmpty()` 判断B+树是否为空。
- `BPlusTree::Insert(const KeyType &key, const ValueType &value, Transaction *transaction = nullptr)` 负责将键值对插入到B+树中去。该函数在实现时会调用实现叶节点以及中间节点插入的公开方法以完成键值对的插入。
- `BPlusTree::Remove(const KeyType &key, Transaction *transaction = nullptr)` 负责将key对应的键值对从B+树中移除。与插入一眼，同样会调用实现叶节点以及中间节点删除的公开方法以完成键值对的删除。
- `BPlusTree::GetValue(const KeyType &key, std::vector<ValueType> &result, Transaction *transaction = nullptr)` 负责将key对应的value从B+树中拿出来，然后插入到传入的vector的尾部。
- `BPlusTree::INDEXITERATOR_TYPE Begin()` 用于返回B+树迭代器的头部。它会用空的key在B+树中寻找对应的节点。
- `BPlusTree::INDEXITERATOR_TYPE Begin(const KeyType &key)` 用于返回B+树迭代器的对于某个特定key的头部。它会用空的key在B+树中寻找对应的节点。
- `BPlusTree::INDEXITERATOR_TYPE End()` 用于返回B+树迭代器的尾部。它会在寻找到某个叶节点的next_page_ID是空是将该节点返回。
- `BPlusTree::FindLeafPage(const KeyType &key, bool leftMost = false, bool rightmost = false)` 用于搜寻传入的key所在的叶节点。它会调用叶节点和中间节点的 `Lookup` 以及 `ValueAt` 公开方法进行查找。
- `BPlusTree::Destroy()` 用于对B+树进行销毁以释放储存空间。

2.3.5 Index Iterator

在本节中，我需要为B+树索引实现一个迭代器。该迭代器能够将所有的叶结点组织成为一个单向链表，然后沿着特定方向有序遍历叶结点数据页中的每个键值对（这在范围查询时将会被用到）。

该迭代器的公开方法以及运算符重载就不做赘述了，主要介绍一下该迭代器类内部的私有成员变量：

```

1 class IndexIterator {
2 private:
3     BPlusTreeLeafPage<KeyType, ValueType, KeyComparator> *leaf_;
4     BufferPoolManager *buffer_pool_manager_;
5     int index_;
6 };

```

`leaf_` 变量主要负责记录迭代器当前处于哪个叶节点上，`buffer_pool_manager` 则为该B+树的内存池，`index_` 变量主要负责记录迭代器当前处于叶节点数组中的哪个index上。

2.4 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。

2.4.1 相关类的介绍

2.4.1.1 Catalog

CatalogMetaData保存table和index编号和序列化使用的page id的对应关系用于恢复

```

1 private:
2     static constexpr uint32_t CATALOG_METADATA_MAGIC_NUM = 89849;
3     std::map<table_id_t, page_id_t> table_meta_pages_;
4     std::map<index_id_t, page_id_t> index_meta_pages_;

```

CatalogManager保存 table和index的name、id和对象的对应关系，还有该catalog的自己的metadata、buffer pool manager、mem heap。

```

1 [[maybe_unused]] BufferPoolManager *buffer_pool_manager_;
2 [[maybe_unused]] LockManager *lock_manager_;
3 [[maybe_unused]] LogManager *log_manager_;
4 [[maybe_unused]] CatalogMeta *catalog_meta_;
5 [[maybe_unused]] std::atomic<table_id_t> next_table_id_;
6 [[maybe_unused]] std::atomic<index_id_t> next_index_id_;
7 // map for tables
8 std::unordered_map<std::string, table_id_t> table_names_;
9 std::unordered_map<table_id_t, TableInfo *> tables_;
10 // map for indexes: table_name→index_name→indexes
11 [[maybe_unused]] std::unordered_map<std::string,
    std::unordered_map<std::string, index_id_t>> index_names_;

```

```

12  [[maybe_unused]] std::unordered_map<index_id_t, IndexInfo *>
    indexes_;
13  // memory heap
14  MemHeap *heap_;

```

2.4.1.2 table

TableInfo代表一个表，储存tableheap和table的metadata

```

1  private:
2      TableMetadata *table_meta_;
3      TableHeap *table_heap_;
4      MemHeap *heap_; /** store all objects allocated in table_meta
    and table heap */

```

TableMetadata储存表的元信息, table id、table name、table heap的root tag id、还有table的列schema

```

1  private:
2      static constexpr uint32_t TABLE_METADATA_MAGIC_NUM = 344528;
3      table_id_t table_id_;
4      std::string table_name_;
5      page_id_t root_page_id_;
6      Schema *schema_;

```

2.4.1.3 index

IndexInfo代表一个索引，储存索引的B+树，索引的表，索引使用键值，还有自身的metadata

```

1  private:
2      IndexMetadata *meta_data_;
3      Index *index_;
4      TableInfo *table_info_;
5      IndexSchema *key_schema_;
6      MemHeap *heap_;

```

IndexMetadata保存索引元信息key_map_是key_schema _对应列的编号的集合，方便序列化和反序列化

```

1      static constexpr uint32_t INDEX_METADATA_MAGIC_NUM = 344528;
2      index_id_t index_id_;
3      std::string index_name_;
4      table_id_t table_id_;
5      std::vector<uint32_t> key_map_;

```


2.4.2 为上层提供接口

```
1 dberr_t CreateTable(const std::string &table_name, TableSchema
*schema, Transaction *txn, TableInfo *&table_info);
2 dberr_t GetTable(const std::string &table_name, TableInfo
*&table_info);
3 dberr_t GetTables(std::vector<TableInfo *> &tables) const;
4 dberr_t CreateIndex(const std::string &table_name, const
std::string &index_name, const std::vector< std::string >
&index_keys, Transaction *txn, IndexInfo *&index_info);
5 dberr_t GetIndex(const std::string &table_name, const
std::string &index_name, IndexInfo *&index_info) const;
6 dberr_t GetTableIndexes(const std::string &table_name,
std::vector<IndexInfo *> &indexes) const;
7 dberr_t DropTable(const std::string &table_name);
8 dberr_t DropIndex(const std::string &table_name, const
std::string &index_name);
```

GetTable/GetIndex分别是根据name映射关系中找到id再找到对应的对象指针进行返回。

DropTable/DropIndex分别是根据name找到对应指针，将其空间释放并且把map中相关条目删除。

CreatTable根据传入的信息构建TableMetadata，再调用TableHeap的构造函数建立TableHeap，将其组装成为一个表，记录映射再返回指针。

CreatTable根据传入的信息构建IndexMetadata，再调用Index的构造函数建立或者恢复Index的B+树，之后根据tableinfo中的结构取出tuple依次插入index。最后相同的记录映射关系再返回指针。

2.4.3 保存和恢复

- 保存：在catalog被析构的时候，将index table 的元信息序列化，序列化方法同上。每一个index或者table分别序列化到一个页中，catalogmetadata记录序列化的页的编号。最后再将catalogmetadata序列化到指定页中。
- 恢复：先从规定好的页中把catalogmetadata反序列化，获得各个index和table的序列化位置，依次将他们的元信息反序列化出来。
- Table恢复：table的元组信息储存在tableheap中，tableheap只要获得root pageid就可以按照链表的查找方式依次找回所有元组。
- Index恢复：index中b+树在构建的时候，根据传入的index_id从index_root _page中找到对应的root_page_id也就是根节点的编号，按照树的查找方式，就可以找回index的所有节点，恢复B+树。

2.5 SQL EXECUTOR

2.5.1 Syntax Tree

由给定框架中实现：

```
1  /**
2   * Syntax node definition used in abstract syntax tree.
3   */
4  struct SyntaxNode {
5      int id_;    /** node id for allocated syntax node, used for
6                  debug */
7      SyntaxNodeType type_; /** syntax node type */
8      int line_no_; /** line number of this syntax node appears in
9                  sql */
10     int col_no_; /** column number of this syntax node appears
11                 in sql */
12     struct SyntaxNode *child_; /** children of this syntax node
13                                */
14     struct SyntaxNode *next_; /** siblings of this syntax node,
15                                linked by a single linked list */
16     char *val_; /** attribute value of this syntax node, use deep
17                 copy */
18 };
19 typedef struct SyntaxNode *pSyntaxNode;
```

2.5.2 ExecuteEngine

在Parser模块调用 `yyparse()`，完成SQL语句解析后，将会得到语法树的根结点 `pSyntaxNode`，将语法树根结点传入执行器 `ExecuteEngine` 后，`ExecuteEngine` 将会根据语法树根结点的类型，分发到对应的执行函数中，以完成不同类型SQL语句的执行。`ExecuteEngine` 的定义如下：

```
1  class ExecuteEngine {
2      ...
3  private:
4      [[maybe_unused]] std::unordered_map<std::string,
5      DBStorageEngine *> dbs_; // all opened databases
6      [[maybe_unused]] std::string current_db_; /** current
7      database */
8  };
```

本节中，需实现 `ExecuteEngine` 中所有执行函数如下：

- `ExecuteEngine::ExecuteCreateDatabase(*ast, *context)`，从语法树中获取创建数据库的名称，从而创建数据库并插入到 `ExecuteEngine` 的 `unordered_map<std::string, DBStorageEngine *> dbs_` 中；

- `ExecuteEngine::ExecuteDropDatabase(*ast, *context)` , 从 `unordered_map<std::string, DBStorageEngine *> dbs_` 移除对应的数据库, 并delete释放;
- `ExecuteEngine::ExecuteShowDatabases(*ast, *context)` , 遍历 `unordered_map<std::string, DBStorageEngine *> dbs_` 输出数据库name;
- `ExecuteEngine::ExecuteUseDatabase(*ast, *context)` , 更新 `ExecuteEngine` 中的 `string current_db_` 为 `use` 的数据库的name;
- `ExecuteEngine::ExecuteShowTables(*ast, *context)` , 获取 `current_db_` , 调用 `catalog_mgr_->GetTables` 函数获取所有 `table` 并输出名称;
- `ExecuteEngine::ExecuteCreateTable(*ast, *context)` , 获取要创建的table的名称等一系列信息 (列的名称、类型、长度、主键等) , 调用 `catalog_mgr_->CreateTable` 函数创建表, 注: 在这个函数中会自动为主键创建索引;
- `ExecuteEngine::ExecuteDropTable(*ast, *context)` , 获取要删除的表的名称, 调用 `catalog_mgr_->DropTable` 函数删除即可;
- `ExecuteEngine::ExecuteShowIndexes(*ast, *context)` , 调用 `catalog_mgr_->GetTableIndexes` 函数获取所有的索引, 输出相关信息即可;
- `ExecuteEngine::ExecuteCreateIndex(*ast, *context)` , 获取要创建索引的一系列信息 (索引名称、建立表的名称、索引列的属性) , 判断索引列属性是否 `unique` , 调用 `catalog_mgr_->CreateIndex` 函数创建索引;
- `ExecuteEngine::ExecuteDropIndex(*ast, *context)` , 获取要删除索引的名称, 遍历当前的每一个表, 调用 `catalog_mgr_->DropIndex` 函数删除;
- `ExecuteEngine::ExecuteSelect(*ast, *context)`

1. 获取要 `select` 的表和列的名称以及判断条件, 为方便起见, 我们为判断条件创建了一个结构体, 具体如下:

```

1 struct SelectCondition
2 {
3     string attri_name;
4     uint32_t type_; //0等于; 1不等于; 2小于; 3大于; 4小于等于; 5大于等于
5     TypeId type_id_;
6     union Val {
7         float float_;
8         int int_;
9         char *chars_;
10    } value_;
11 };

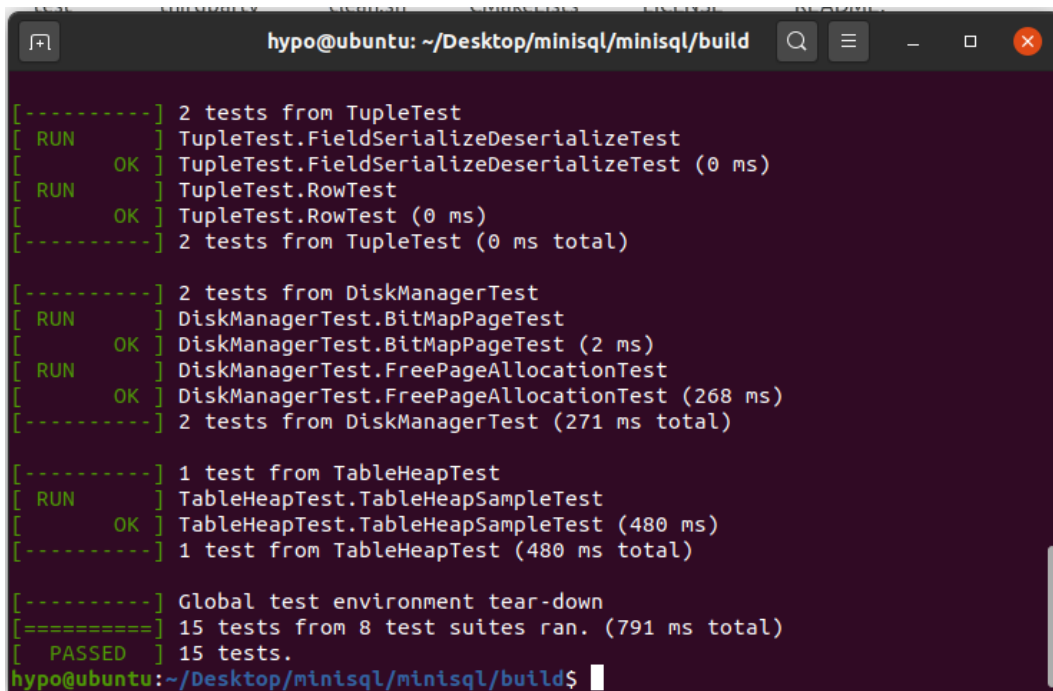
```

2. 若判断条件有多个, 直接迭代器遍历 `TableHeap` 对每一个 `Row` 判断是否符合条件, 若符合则输出相应查询的列的信息;
3. 若判断条件只有一个, 遍历索引, 若:
 - a. 判断条件契合索引, 则利用相应索引的的迭代器进行查询;
 - b. 判断条件不符合索引, 则同2.
- `ExecuteEngine::ExecuteInsert(*ast, *context)`

1. 获取要插入的表;
 2. 获取要插入的每个 `column` 对应的值, 创建 `Field` 数组, 这个过程中需要判断主键约束和unique约束, 可通过遍历 `TableHeap` 实现;
 3. 利用 `Field` 数组创建 `Row`, 调用 `table_heap->InsertTuple` 函数将 `Row` 插入堆表;
 4. 根据 `Row` 创建索引对应的 `key_row` 并获取 `RowId`, 调用 `index->InsertEntry` 函数插入进索引的B+树中。
- `ExecuteEngine::ExecuteDelete(*ast, *context)`
 1. 获取表的名称和删除条件, 删除条件和选择条件一样用同样的结构体记录;
 2. 遍历堆表, 若记录满足删除条件, 则调用 `table_heap->ApplyDelete` 函数删除记录;
 3. 遍历索引, 调用 `RemoveEntry` 函数删除相应索引。
 - `ExecuteEngine::ExecuteUpdate(*ast, *context)`
 1. 获取表的名称和更新条件, 更新条件和选择条件一样用同样的结构体记录;
 2. 遍历堆表, 若记录满足条件, 则调用 `table_heap->UpdateTuple` 函数更新记录;
 3. 遍历索引, 调用 `RemoveEntry` 先删除记录, 再调用 `InsertEntry` 插入新记录。
 - `ExecuteEngine::ExecuteExecfile(*ast, *context)`, 循环文件读入, 执行语句即可。

3 功能演示

首先, 为了检验代码的有效性, 我们编译并运行 `minisql_test` 文件, 得到结果如下图所示:



```
hypo@ubuntu: ~/Desktop/minisql/minisql/build
[-----] 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 (268 ms)
[-----] 2 tests from DiskManagerTest (271 ms total)

[-----] 1 test from TableHeapTest
[ RUN    ] TableHeapTest.TableHeapSampleTest
[ OK     ] TableHeapTest.TableHeapSampleTest (480 ms)
[-----] 1 test from TableHeapTest (480 ms total)

[-----] Global test environment tear-down
[=====] 15 tests from 8 test suites ran. (791 ms total)
[ PASSED ] 15 tests.
hypo@ubuntu:~/Desktop/minisql/minisql/build$
```

可以看到, 所有的test测试点都已经通过, 表明我们的代码在有效性上是可以保证的。接下来运行 `main` 程序, 检查我们的程序是否能对各条指令做出正确的反应。

首先我们进行数据库的创建尝试。在运行了下述三条创建数据库的指令后, 输入显示数据库的指令, 可以看到三个数据库均被成功创建。

```
minisql > create database db0;
EXECUTE TIME: 0.006134
minisql > create database db1;
EXECUTE TIME: 0.006298
minisql > create database db2;
EXECUTE TIME: 0.0065
minisql > show databases;
db2
db1
db0
EXECUTE TIME: 3.4e-05
minisql >
```

接下来我们进行表的创建。可以看见，在进行插入表操作后输入显示表的指令，可以看到该表已经被成功创建。

```
minisql > use db0;
create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
show tables;EXECUTE TIME: 3.5e-05
minisql > EXECUTE TIME: 0.000336
minisql > show tables;
account
```

接下来我们对表进行数据的插入。执行 `execfile account00.txt` 指令，在该file中我们存有大数量的插入指令。从下图可以看到，我们的插入操作执行成功。

```
1 insert into account values(12500000, "name0", 514.35);
2 insert into account values(12500001, "name1", 103.14);
3 insert into account values(12500002, "name2", 981.86);
4 insert into account values(12500003, "name3", 926.51);
5 insert into account values(12500004, "name4", 4.87);
6 insert into account values(12500005, "name5", 437.08);
7 insert into account values(12500006, "name6", 373.75);
8 insert into account values(12500007, "name7", 681.87);
9 insert into account values(12500008, "name8", 666.64);
10 insert into account values(12500009, "name9", 67.46);
11 insert into account values(12500010, "name10", 742.09);
12 insert into account values(12500011, "name11", 539.08);
13 insert into account values(12500012, "name12", 595.48);
14 insert into account values(12500013, "name13", 526.95);
15 insert into account values(12500014, "name14", 562.35);
16 insert into account values(12500015, "name15", 277.79);
17 insert into account values(12500016, "name16", 153.52);
18 insert into account values(12500017, "name17", 636.65);
19 insert into account values(12500018, "name18", 286.89);
20 insert into account values(12500019, "name19", 632.82);
21 insert into account values(12500020, "name20", 300.35);
```

```
hypo@ubuntu: ~/Desktop/minisql/minisql/build
第i行: 9980
第i行: 9981
第i行: 9982
第i行: 9983
第i行: 9984
第i行: 9985
第i行: 9986
第i行: 9987
第i行: 9988
第i行: 9989
第i行: 9990
第i行: 9991
第i行: 9992
第i行: 9993
第i行: 9994
第i行: 9995
第i行: 9996
第i行: 9997
第i行: 9998
第i行: 9999
第i行: 10000
第i行: 10001
EXECUTE TIME: 3.84672
minisql >
```

接下来我们将通过select语句对插入的结果进行验证，同时也初步检验select语句的性能。执行 `select * from account;` 语句，可以看到一共查询到一万条记录，这与我们此前插入的数量吻合。

```
12509984      name9984      6.04
12509985      name9985      784.25
12509986      name9986      225.93
12509987      name9987      150.72
12509988      name9988      39.15
12509989      name9989      893.13
12509990      name9990      465.56
12509991      name9991      729.14
12509992      name9992      660.28
12509993      name9993      588.19
12509994      name9994      340.82
12509995      name9995      181.26
12509996      name9996      788.87
12509997      name9997      119.38
12509998      name9998      623.28
12509999      name9999      86.27
.....
一共查到 10000条记录!
EXECUTE TIME: 0.180029
minisql >
```

我们再对select语句进行进一步的检验。执行如下语句段：

```
1 select * from account where id = 12509996;
2 select * from account where balance = 788.87;
3 select * from account where id <> 12509996;
4 select name, balance from account where balance >= 100.00 and
   balance <= 200.00;
```

先执行两条等值查询语句，一个是主键一个是非主键，可以看到我们的程序均返回了正确的结果。值得注意的是，利用主键进行的查询比未用主键的快了很多倍，这是因为主键上有主键索引，因此查询速度会比普通的select快几个数量级。

```

minisql > select * from account where id = 12509996;
可利用索引: prim_index进行优化查询
selectWithIndex 12509996
12509996      name9996      788.87
一共查到 1条记录!
EXECUTE TIME: 0.000264
minisql > select * from account where balance = 788.87;
12509996      name9996      788.87
.....
一共查到 1条记录!
EXECUTE TIME: 0.051712
minisql >

```

再执行第三条不等值查询语句，可以看到返回了9999条记录，说明我们此处的逻辑设计没有问题。

```

12509990      name9990      465.56
12509991      name9991      729.14
12509992      name9992      660.28
12509993      name9993      588.19
12509994      name9994      340.82
12509995      name9995      181.26
12509997      name9997      119.38
12509998      name9998      623.28
12509999      name9999      86.27
一共查到 9999条记录!
EXECUTE TIME: 0.177652
minisql >

```

再执行第四句双限制条件的select特定属性语句，可以看到我们的程序跑出了正确的结果。

```

name9890      162.02
name9891      155.44
name9894      161.31
name9903      157.48
name9910      131.12
name9978      150.97
name9987      150.72
name9995      181.26
name9997      119.38
.....
一共查到 1031条记录!
EXECUTE TIME: 0.074374
minisql >

```

经过上述select语句的执行，我们大致可以检验出select部分的逻辑没有问题。接下来我们验证一下主键和unique的约束性。

```

1 insert into account values(12500000, "namename", 111.01);
2 insert into account values(125, "name9997", 111.11);

```

```

minisql > insert into account values(12500000, "namename", 111.01);
PRIMARY KEY约束冲突
EXECUTE_FAILED!!!!
EXECUTE TIME: 5e-05
minisql > insert into account values(125, "name9997", 111.11);
UNIQUE约束冲突
EXECUTE_FAILED!!!!
EXECUTE TIME: 5.6e-05
minisql >

```

可以看到由于id是主键，name设置为了unique，所以我们的insert操作没有成功。接下来我们开始检验index部分的指令是否存在问题。


```

1 create index idx01 on account(name);
2 select * from account where name = "name9996";
3 drop index idx01;
4 select * from account where name = "name9996";

```

首先执行create index的操作，可以看到我们的index被成功创建，随后的select语句可以看出此前那表中的数据也都被插入到了index中去。随后，我们执行对index的drop操作，并再次对同一组数据进行查询。可以看到，查询依然成功，这表明删除index成功的时候里面的数据不会受到影响。

```

minisql > create index idx01 on account(name);
EXECUTE TIME: 7.98959
minisql > select * from account where name = "name9996";
可利用索引: idx01进行优化查询
12509996          name9996          788.87
一共查到 1条记录!
EXECUTE TIME: 0.000268
minisql > drop index idx01;
删除索引名称: idx01
EXECUTE TIME: 9.2e-05
minisql > select * from account where name = "name9996";
12509996          name9996          788.87
.....
一共查到 1条记录!
EXECUTE TIME: 0.076191
minisql >

```

值得注意的是，在drop了index之后，我们的查询速度变慢了几个数量级。这足以说明利用B+树实现的index可以优化改进查找速度。

接下来检验delete和update语句的正确性。

```

1 delete from account where name = "name9996";
2 select * from account where name = "name9996";
3 update account set balance = 111.11 where id ≥ 12509900 and id
  < 12509920;
4 select * from account where id ≥ 12509900 and id < 12509920;

```

在首先进行delete之后，我们可以看到此前找得到的该条记录已经消失了。这说明我们的delete操作已经成功。

```

minisql > delete from account where name = "name9996";
EXECUTE TIME: 0.075329
minisql > select * from account where name = "name9996";
.....
一共查到 0条记录!
EXECUTE TIME: 0.065994
minisql >

```

然后，我们再对数据进行update操作，随后再进行select，我们可以看到查询出的数据的balance属性都变成了111.11。这说明我们的update操作已经成功。

```

12509909      name9909      111.11
12509910      name9910      111.11
12509911      name9911      111.11
12509912      name9912      111.11
12509913      name9913      111.11
12509914      name9914      111.11
12509915      name9915      111.11
12509916      name9916      111.11
12509917      name9917      111.11
12509918      name9918      111.11
12509919      name9919      111.11
.....
一共查到 20条记录!
EXECUTE TIME: 0.062783
minisql >

```

我们再来检验数据库在退出之后是否可以成功恢复：

```

1 quit;
2 show databases;
3 use db0;
4 select * from account;

```

执行quit之后再进入main程序，再对数据库进行显示，我们可以看到此前创建的三个数据库都还存在。

```

minisql > quit;
EXECUTE TIME: 0.001624
bye!
hypo@ubuntu:~/Desktop/minisql/minisql/build$ ./bin/main
minisql > show databases;
db2
db1
db0
EXECUTE TIME: 0.694659
minisql > use db0;
EXECUTE TIME: 1.7e-05
minisql >

```

然后再对表中的全部数据进行select操作，可以看到除去我们此前删去的一条数据之外，其他9999条数据均被成功恢复。

```

12509990      name9990      465.56
12509991      name9991      729.14
12509992      name9992      660.28
12509993      name9993      588.19
12509994      name9994      340.82
12509995      name9995      181.26
12509997      name9997      119.38
12509998      name9998      623.28
12509999      name9999      86.27
.....
一共查到 9999条记录!
EXECUTE TIME: 0.207534
minisql >

```

最后我们再检验drop操作的正确性。使用drop命令删除表，再展示数据库中所有的表，我们可以发现表已经消失，这说明我们的删除操作已经成功。

```

minisql > drop table account;
EXECUTE TIME: 9.3e-05
minisql > show tables;
EXECUTE TIME: 2.8e-05
minisql >

```


4 优化分析

4.1 Select with Index

在执行 `select` 语句时，针对不同的情况，我们有遍历堆表查询和调用索引查询两种策略。分析和比较这两种策略：

- 对于第一种策略，需要调用堆表的迭代器遍历整个堆表，也即双向链表，不考虑其中的复杂操作，单条查询的时间复杂度为 $O(N)$ ；
- 对于第二种策略，实际操作相当于是B+树的查询，在获取 `RowId` 后即可定位要查询的记录在堆表中位置，从而单条查询的时间复杂度为 $O(\log N)$ 。

总体来说，当表中记录达到一定规模，采取第二种策略相较于第一种策略有很大的提升，以下为实际测试的效果图：

创建索引前，普通单条查询，平均时间为 `0.15s` 左右：

```
EXECUTE TIME: 0.140625
minisql > select * from account where name = "name24563";
3304563          name24563          335.84
.....
一共查到 1条记录!
EXECUTE TIME: 0.140625
minisql > select * from account where name = "name24563";
3304563          name24563          335.84
.....
一共查到 1条记录!
EXECUTE TIME: 0.15625
minisql > select * from account where name = "name24563";
3304563          name24563          335.84
.....
一共查到 1条记录!
EXECUTE TIME: 0.15625
minisql > select * from account where name = "name24563";
3304563          name24563          335.84
.....
一共查到 1条记录!
EXECUTE TIME: 0.15625
minisql > select * from account where name = "name24563";
3304563          name24563          335.84
.....
一共查到 1条记录!
```

创建索引后，索引查询，可见基本不用时间即可查询到结果：

```
minisql > select * from account where name = "name24563";
可利用索引: id1进行优化查询
3304563          name24563          335.84
一共查到 1条记录!
EXECUTE TIME: 0
minisql > select * from account where name = "name24563";
可利用索引: id1进行优化查询
3304563          name24563          335.84
一共查到 1条记录!
EXECUTE TIME: 0
```

4.2 Memheap(bonus)

4.2.1 描述

Bonus: `MemHeap` 通常用于内存分配和回收的管理，它能在析构时自动释放分配的内存空间，在一定程度上能够避免内存泄漏的问题。同时，合理的内存分配方式能够避免多次频繁调用 `malloc` 对性能造成的影响。本实验框架中仅给出了

`SimpleMemHeap` 用于简单的内存分配和回收。若实现一种新的内存分配和管理方式（继承 `MemHeap` 类实现其分配和回收函数）提高内存分配和回收的性能可以获得一定的加分。

4.2.2 具体实现

新的Memheap采用预先分配好大量内存并用bitmap的方式来管理内存的分配和回收的方式来优化。

采用内存池的方式，具体有 `{4, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128}` 17种大小的内存块，每个内存块有 `BLOCK_NUM` 的数量，对于每种内存池，采用 `Bitmap` 的如下结构来进行管理，`bool bitmap[BLOCK_NUM]` 的每一位的 `0/1` 代表了对应内存块的状态“空闲/占用”，同时记录当前第一个空闲块的bit位 `firstzerobit` 和当前总的空闲块的数量 `freespaceSize`。`Bitmap` 可通过调用函数 `ChangeFreeNext()` 和 `ResetZero(int bit)` 来分配和归还内存。

```
1  #define BLOCK_NUM 1600 // 每个内存池block的数量
2  #define MEMPPOOL_NUM 17 // 内存池的种类数量 (4, 8, 16, 24, 32,
   40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128)
3  #define MEMPPOOLS_SIZE (BLOCK_NUM * 1092)
4
5  struct Bitmap
6  {
7      void ChangeFreeNext()
8      {
9          bitmap[ firstzerobit ] = 1;        // 更新当前freebit to 1
10         if (--freespaceSize)
11         {
12             while ( bitmap[(++firstzerobit) % BLOCK_NUM] ) {}
13         }
14     }
15     void ResetZero(int bit)
16     {
17         bitmap[bit] = 0;
18         freespaceSize++;
19     }
20     bool bitmap[BLOCK_NUM] = {0};
21     int firstzerobit = 0, freespaceSize = BLOCK_NUM;
22 };
23
24 class MyMemPools : public MemHeap {
```

```

25 public:
26     MyMemPools()
27     {
28         head_pointer_ = malloc(MEMP00LS_SIZE); // 分配总空间
29     }
30     ~MyMemPools()
31     {
32         free(head_pointer_); //释放全部内存
33     }
34
35     void *Allocate(size_t size)
36     {
37         if (size ≤ 0)
38             return nullptr;
39         if (size > 128)
40         {
41             return malloc(size);
42         }
43
44         size_t pool_index = pool_map[size]; // 根据分配空间的大小获取
        对应内存池的下标
45         if (!bmap[pool_index].freespacesize) // 当前内存池无空
        闲block
46         {
47             return malloc(size);
48         }
49
50         auto block_pointer = (uint8_t *)head_pointer_ +
        head_block_offset[pool_index] + block_size[pool_index] *
        bmap[pool_index].firstzerobit; // 获取要分配给的block
51
52         bmap[pool_index].ChangeFreeNext(); //修改block对应bit为1,同时
        更新第一空闲位
53
54         return block_pointer;
55     }
56
57     void Free(void *ptr, size_t deallocate_size)
58     {
59         if (ptr < head_pointer_ || ptr ≥ (uint8_t *)head_pointer_
        + MEMP00LS_SIZE) // 要释放的指针是new分配的
60         {
61             free(ptr);
62         }
63         else
64         {
65             size_t pool_index = pool_map[deallocate_size];
66             // memset(del_ptr, 0, deallocate_size); //清空block
67
68             // 重新将bit位置0:

```

```

69         int del_bit = ( (uint8_t *)ptr - (uint8_t
*)head_pointer_ - head_block_offset[pool_index] ) /
    block_size[pool_index];
70         bmap[pool_index].ResetZero(del_bit);
71     }
72 }
73
74 private:
75     void *head_pointer_;           // 所有内
    存的首地址
76
77     struct Bitmap bmap[MEMPOOL_NUM];
78
79     const int block_size[MEMPOOL_NUM] = {4, 8, 16, 24, 32, 40,
48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128};
80
81     const int head_block_offset[MEMPOOL_NUM] = {0, 6400,
19200, 44800, 83200, 134400, 198400, 275200, 364800, 467200,
582400, 710400, 851200, 1004800, 1171200, 1350400, 1542400};
82
83     const size_t pool_map[129] = {0,
84                                     0, 0, 0, 0,
85                                     1, 1, 1, 1,
86                                     2, 2, 2, 2, 2, 2, 2, 2,
87                                     3, 3, 3, 3, 3, 3, 3, 3,
88                                     4, 4, 4, 4, 4, 4, 4, 4,
89                                     5, 5, 5, 5, 5, 5, 5, 5,
90                                     6, 6, 6, 6, 6, 6, 6, 6,
91                                     7, 7, 7, 7, 7, 7, 7, 7,
92                                     8, 8, 8, 8, 8, 8, 8, 8,
93                                     9, 9, 9, 9, 9, 9, 9, 9,
94                                     10, 10, 10, 10, 10, 10, 10, 10,
95                                     11, 11, 11, 11, 11, 11, 11, 11,
96                                     12, 12, 12, 12, 12, 12, 12, 12,
97                                     13, 13, 13, 13, 13, 13, 13, 13,
98                                     14, 14, 14, 14, 14, 14, 14, 14,
99                                     15, 15, 15, 15, 15, 15, 15, 15,
100                                    16, 16, 16, 16, 16, 16, 16, 16};
101 };

```

5 项目总结

历经一个夏学期的时间，在老师和助教的带领和指导下，我们成功地完成了Minisql的实现。

从一开始的茫然无知和对整个框架复杂程度的感叹，到如今可以对每个模块侃侃而谈，对他们之间的联系和相互作用关系了如指掌，我们诚然是经历了很多的。

“上兵伐谋，其次伐交，其次伐兵，其下攻城。”在整个项目的开始，在初步了解了整个工程的布局后，我们对之后每周的进度也有了初步的谋划。在保证项目的推进和代码的工整以及可读性的前提下，我们尽量做到游刃有余。之后八周，所遇挫折不少，挑灯夜战也是常态。有令人拍手叫绝之处，更不乏扼腕叹息之时。坦言，这次的Minisql是前所未有之难，但幸而，我组成员皆善战好胜之辈，虽苦海而渡之，虽漫路而索之，时至今日，无愧于心矣。

“Good design demands good compromises.”时间紧迫，对于整个项目的各个部分，我们有所牺牲，有所妥协，然遇艰深之处，亦有所坚持，有所顿悟。

摆脱了以往低效的合作方式，我们采用的 `git` 的代码协作方式，一步步走来，全留脚印；一行行commit，皆是成长；一次次push，尽为蜕变。正所谓磨刀不误砍柴工，前人的智慧须好好领悟和利用。

过去在风中，未来亦从容。这次实验顺利完成，待老师和助教批阅！