MiniSQL 个人详细报告

张峻瑜

时间：2022 年 6 月 4 日

Contents

1 个人分工说明

本人在本项目中主要负责如下几个部分：DiskManager 的实现，BufferPoolManager 的实现以及 LRU Replacer 的优化、Clock replacer 的实现、Index 的实现并重构了 Index 模块，抛弃了原有模板的设计、CatalogManager 的实现、内存管理方式上的优化以及 ManagedHeap 的实现。

2 我完成的工作

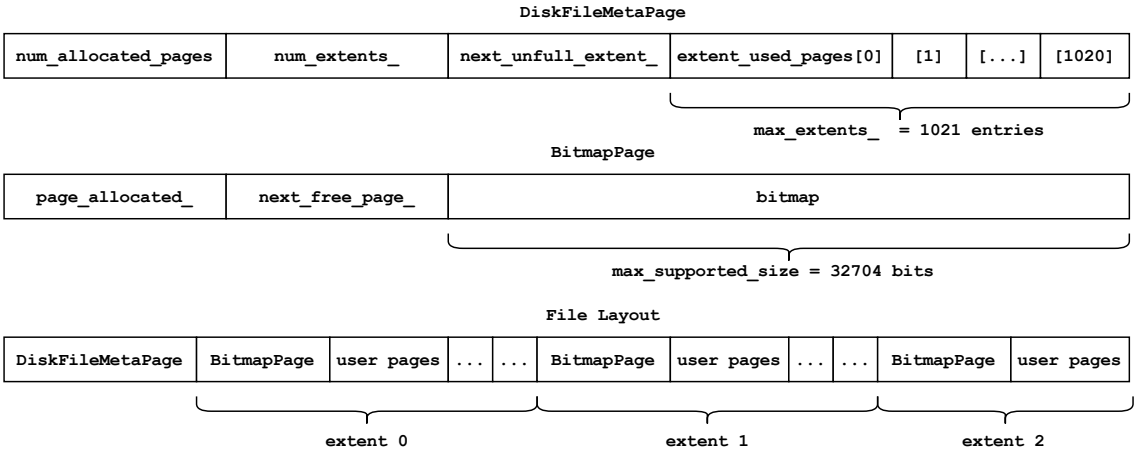
2.1 DiskManager 模块

本项目中，每个数据库的数据各自存储在单一文件之中，文件名为 {数据库名}.db。该文件被划分为若干个页 (`page`), 每页大小为 4096 字节。每个页被物理页号 (`physical_page_id`) 唯一标识, 同时其中的用户页具有供上层模块标识用的逻辑页号 (`logical_page_id`)。页的分配、释放、读取与写入由 DiskManager 模块统一管理。

- 用户页: 上层模块通过 DiskManager 申请的页。
- 系统页: 除过用户页之外的页。这些页被 DiskManager 用于存储关于磁盘、页管理的原信息。

- 物理页号：DiskManager 从磁盘写入/读出页时使用的页号，物理页号与页一一对应。
- 逻辑页号：为了隐藏 DiskManager 的实现细节，系统页对上层模块不可见，只有用户页能被上层模块访问。每个用户页被逻辑页号唯一标识。在用户页中，逻辑页号从 0 开始连续增长。

.db 文件的文件布局如下：



为了加速页的分配与删除，本项目不仅仅对用户页采用了缓存，也对系统页采用了缓存。系统页的缓存由 DiskManager 单独管理，管理的函数为：

- `ReadPhysicalMetaPage(page_id_t physical_page_id)`
  1. 从 buffer 中寻找该页
  2. 若找到，则直接从内存中读取
  3. 若未找到，则从磁盘中读取，并通过 `replacer_` 替换掉 buffer 中的某页。
  4. 若被替换的页为 dirty，则把该页写入磁盘
- `WritePhysicalMetaPage(page_id_t physical_page_id)`
  1. 从 buffer 中寻找该页
  2. 若未找到，将该页存入 buffer，写入数据，并标记为 dirty
  3. 若找到，直接写入数据，并标记为 dirty

`DiskManager` 模块的接口以及实现如下：

- `void ReadPage(page_id_t logical_page_id, char *page_data)`：从文件中读取 `logical_page_id` 标识的用户页，并把页数据写入 page_data 指向的内存中。

1. 调用 IsPageFree()，若该页未被分配，抛出错误。
2. 若该页未被分配,调用 ReadPhysicalPage() 读取数据。ReadPhysicalPage 的实现如下：
   (a) 调用 MapPageId(), 根据逻辑页号计算出 extent_id 对应的位图页与物理页号
   (b) 调用 ReadPhysicalMetaPage()，读取位图页
   (c) 若位图页中该页对应的 bit 为 0，抛出异常
   (d) 否则，将该页的数据读入内存

- void WritePage(page_id_t logical_page_id, const char *page_data)：将 page_data 指向的 4096 字节写入 logical_page_id 所标识的用户页对应的文件块中。

  1. 调用 IsPageFree()，若该页未被分配，抛出错误。
  2. 调用 MapPageId(), 根据 logical_page_id 计算 physical_page_id。
  3. 调用 ReadPhysicalPage()，将数据读入内存。

- page_id_t AllocatePage()：申请分配一个用户页，返回新分配用户页的逻辑页号。

  1. 根据 next_extent_id_,找到对应 extent 的位图页。若大于 GetMaxExtentNum()，抛出异常。
  2. 从位图页中获取 next_free_page，若大于 GetMaxSupportedSize()，抛出异常。
  3. 调用 BitmapPage::AllocatePage，分配一个页，并保存分配的页号。
  4. 更新 BitmapPage::next_free_page_。
  5. 更新 next_unfull_extent_。
  6. 返回分配的页号。

- void DeAllocatePage(page_id_t logical_page_id)：释放被 logical_page_id 所标识的用户页。

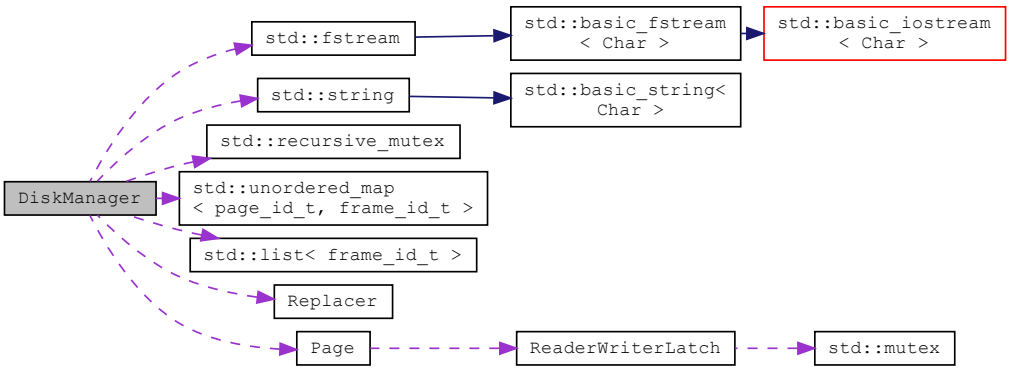
  1. 调用 IsPageFree，若未被分配，抛出错误。
  2. 调用 ReadPhysicalMetaPage，读入位图页。
  3. 调用 BitmapPage::DeAllocatePage，释放页，若失败则抛出错误。
  4. 更新 BitmapPage::next_free_page_。
  5. 更新 next_unfull_extent_。

- bool IsPageFree(page_id_t logical_page_id)：查询被 logical_page_id 所标识的用户页是否已经被分配。

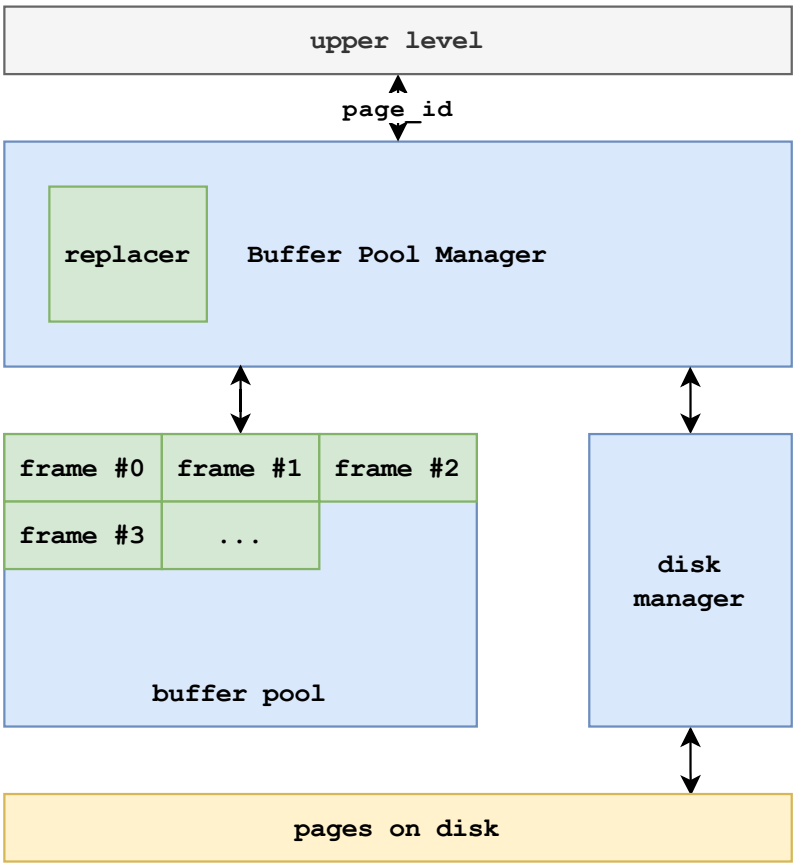
1. 调用 `FetchPhyscicalMetaPage`，获取对应的位图页。
2. 根据位图页中对应的位，返回分配情况。

- `void Close()`：关闭 DiskManager, 保存相关信息，释放文件句柄。
  1. 将全部 buffer 中的 dirty 页写入磁盘。
  2. 将 `DiskMetaPage` 写入磁盘。
  3. 释放 buffer 的内存。

该模块的依赖关系图如下：

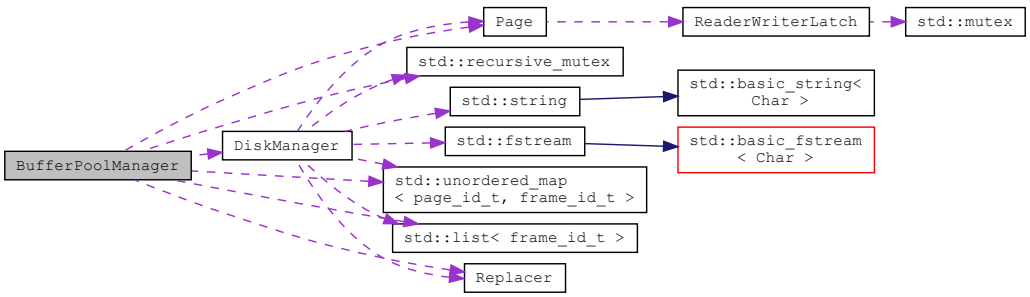


2.2 BufferPoolManager 模块



为了提升从磁盘读/写页的速度，BufferPoolManager 对所有用户页进行了缓存处理，这大大加快了上层模块获取/写入页数据的速度。

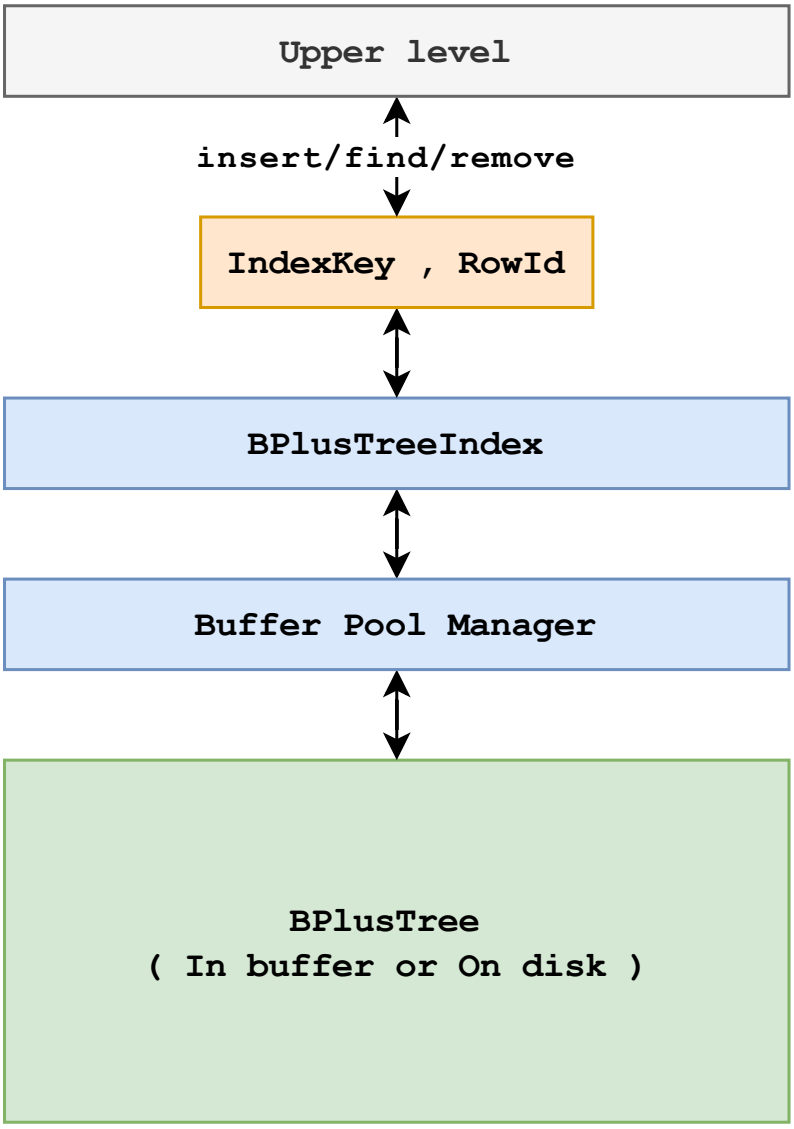
该模块的依赖关系图如下：



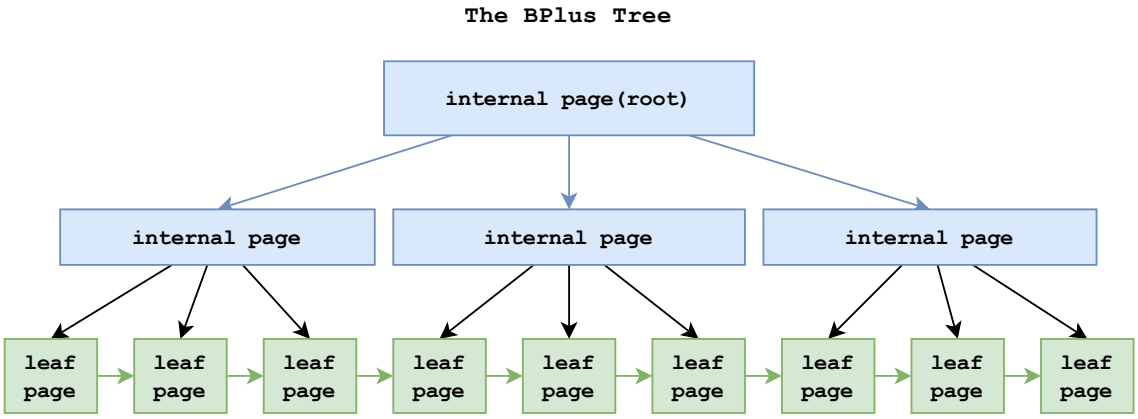
该模块的接口及实现如下：

- `Page * FetchPage(page_id_t LogicalPageId)`：获取 LogicalPageId 标识的用户页对应的 Page 对象。
  1. 检查该页是否在缓存中。
  2. 若不在，从 free_list 中获取或调用 replacer 获取要替代的缓存页，并把目标页读入该缓存页。
  3. 目标页引用计数加一。
  4. 返回目标页对应的 Page 对象。
- `bool UnPinPage(page_id_t LogicalPageId,bool dirty)`：引用计数减一。
- `page_id_t NewPage(page_id_t & pageId)`：申请分配一个用户页。
  1. 调用 `DiskManager::AllocatePage`，申请分配一个新的用户页。
  2. 将该页加入缓存，引用计数加一。
- `bool DeletePage(page_id_t pageId)`：删除一个用户页。
  1. 检查该页是否在缓存中。
  2. 若在，从缓存和 replacer 中移除该页，将对应的 frame 加入 free_list
  3. 调用 `DiskManager::DeAllocatePage`，释放该页。
- `FlushPage(page_id_t pageId)`：手动将该页写入磁盘。
- `FlushAll`：将全部为 dirty 的页写出磁盘。
- `IsPageFree`：查询该页是否已被分配。

2.3 IndexManager 模块



IndexManager 负责管理索引项的添加、删除与访问。由于索引本身特殊的数据结构，给定某个键，我们能够快速的从索引中获取该键对应记录的 RowId，从而快速从堆表中取得该行记录。本项目实现了在**任意长度**键上建立几乎**零额外空间消耗**（也就是，除了存储键本身的值与 RowId 所需的空间之外，几乎没有额外的空间消耗）的 B+ 树索引。在一个表的一个索引之中，包含了若干个索引项。每个索引项由两部分构成：IndexKey 与 RowId。IndexKey 存储了键值，RowId 存储了该索引项对应记录在堆表中的位置。本模块的功能就是实现建立 B+ 树索引、删除、添加、访问索引项的功能。



一颗 B+ 树由若干个叶子节点 (BPlusTreeLeafPage) 与内部节点 (BPlusTreeInternalPage) 构成。内部节点按照增序存储了若干个子树的最小键值，以及子树的根节点页号。索引项以增序存储在叶子节点之中。每个内部节点或者叶子节点都单独占用一整个用户页。对于不同的索引，其 IndexKey 所占空间不同，因此每个内部节点与叶子节点所能存储的最大项数也不同，且在程序运行期间动态计算决定。以下是 IndexKey、BPlusTreePage、BPlusTreeLeafPage 与 BPlusTreeInternalPage 的内存布局。

IndexKey

key_size_	value_ (variable length of key data ...)

BPlusTreePage(the header part)

page_type_	lsn_	key_size_	size_	max_size_	parent_page_id_	page_id_	...

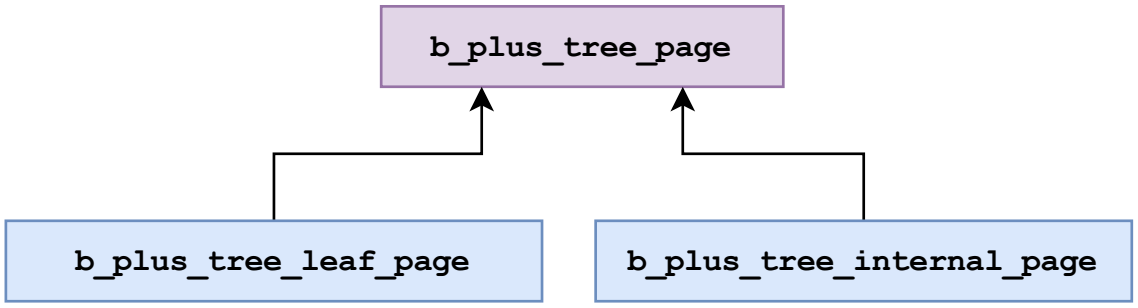
BPlusTreeLeafPage

header	next_page_id_	IndexKey 0	RowId 0	IndexKey 1	RowId 1	...

BPlusTreeInternalPage

header	IndexKey 0	page_id_t 0	IndexKey 1	page_id_t 1	...

BPlusTreePage、BPlusTreeLeafPage 与 BPlusTreeInternalPage 的继承关系如下：



值得注意的是，本模块摒弃了原框架中的模板设计，而是采用了单一的类来实现 BPlusTreeIndex，该实现方法有诸多好处，详见” 设计亮点与 Bonus” 章节。

本项目未采用 GenericKey 以及 GenericComparator，而是使用自己实现的 IndexKeyComparator 来进行 IndexKey 之间的比较。原理与 GenericComparator 相同，但在内存管理方式上略有区别，详见” 设计亮点与 Bonus” 章节。

本模块提供的接口以及实现如下：

- `BPlusTreeIndex::InsertEntry(const Row &key,RowId rowid)`：插入一个索引项。
  1. 将 row 序列化为 Indexkey.
  2. 调用 `BPlusTree::Insert` 函数，尝试插入该 key。
  3. 若插入失败，则代表已有重复键值，返回 False.
  4. 否则，返回 true.
  5. `BPlusTree::Insert` 函数实现如下：
     (a) 调用 `BPlusTree::InternalInsert()`，插入键值，并获取可能的“分裂页”。分裂页指的是，插入子节点导致节点分裂而产生的新页。
     (b) 若有重复，返回 false。否则：
     (c) 若有分裂页，创建新的树根，并将两个子树连接到树根上。
     (d) 返回 true.
     (e) `BPlusTree::InternalInsert` 函数实现如下：
        i. 若该页为叶子节点，执行 2。否则执行 6。
        ii. 若有重复，则返回：重复。
        iii. 将 IndexKey 与 RowId 组合成 BLeafEntry，并二分查找，插入该项。
        iv. 若页已满，则申请新页，并对半分配两页所含的索引键值。
        v. 返回：节点中第一个键值、新分配的页、无重复。
        vi. 二分查找，找到要插入索引项应当处于的子节点 x，调用 `InternalInsert(x)`。
        vii. 若有重复，则返回：重复。
        viii. 若无新创建的页，则返回：节点中第一个键值、无重复。
        ix. 否则，将新页插入该节点。
        x. 若页已满，则申请新页，并对半分配两页所含的索引键值。
        xi. 返回：节点中第一个键值、新分配的页、无重复。
- `BPlusTreeIndex::RemoveEntry(const Row &key)`：删除一个索引项。
  1. 将 row 序列化为 Indexkey。
  2. 调用 `BPlusTree::Remove`。

3. `BPlusTree::Remove` 的实现如下：
4. (a) 调用 `BPlusTree::InternalRemove()`，移除索引项。
   (b) 若返回未找到，则返回 false。
   (c) 若某个子节点的子节点数变为 0，则删除该子节点。
   (d) 若仅有一个子节点，则删除根节点，并把子节点作为新的根节点。
   (e) `BPlusTree::InternalRemove()` 的实现如下：
      i. 若该节点为叶子节点，执行 2。否则，执行 4。
      ii. 二分查找，尝试删除。
      iii. 若未找到，返回错误，否则，返回该节点的 size 与第一项的键值。
      iv. 该节点为内部节点，二分查找找到要下一步搜索子节点。
      v. 调用 `BPlusTree::InternalRemove()`。
      vi. 若未找到，返回错误。
      vii. 若子节点的 size 小于 `GetMinSize()`，尝试将子节点与其伙伴节点合并，或者从其伙伴节点中调换一个索引项到子节点中。
      viii. 返回该节点的 size 与第一项的键值。

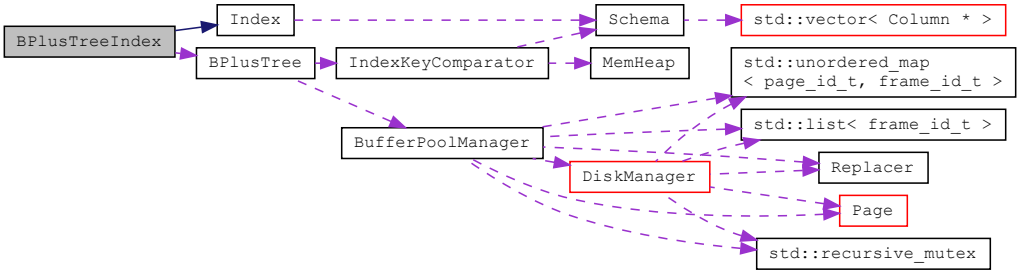
- `BPlusTreeIndex::Scankey`：读取一个索引项。
  1. 沿着 B+ 树自顶向下递归查找。在每个节点内部，采用二分查找算法。
- `BPlusTreeIndex::GetBeginIterator`：获取首迭代器。该迭代器的键值从最小或指定索引项开始递增，到尾迭代器结束。
  1. 二分查找。
- `BPlusTreeIndex::FindLastSmallerOrEqual(const Row & key)`：获取最后一个小于等于 key 的索引项的迭代器。
  1. 二分查找。
- `BPlusTreeIndex::GetEndIterator`：获取尾迭代器。

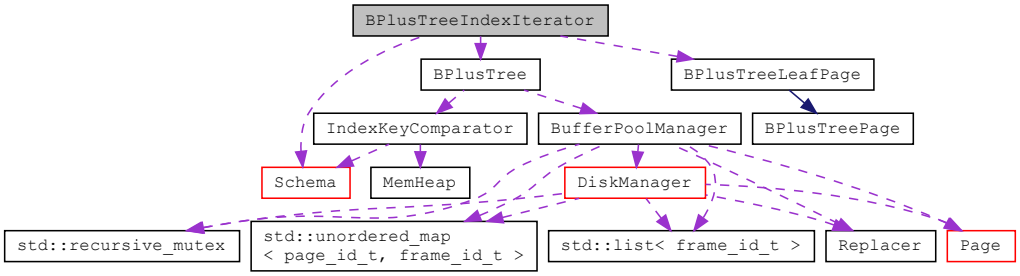
不同于 TableHeap 的迭代器，B+ 树的迭代器本身并不存储任何实际数据。这是由于 B+ 树迭代器解引用之后所得的数据与实际存储在内存中的数据一致，因此无需分配额外的空间。BPlusTreeIterator 解引用之后，返回的就是实际存储在叶子节点之中的索引项。迭代器成员如下：

- BPlusTree* tree：Iterator 所在的 B+ 树。
- BPlusTreeLeafNode *：Iterator 所在的叶子节点。在 Iterator 创建直到 End 或者析构期间，该节点对应的 page 始终被 pin.

- Schema * ：用于 BPlusTreeIteraotr::IsNUll()，判断该迭代器对应的索引项键值是否为空。

类关系图如下：





2.4 CatalogManager 模块

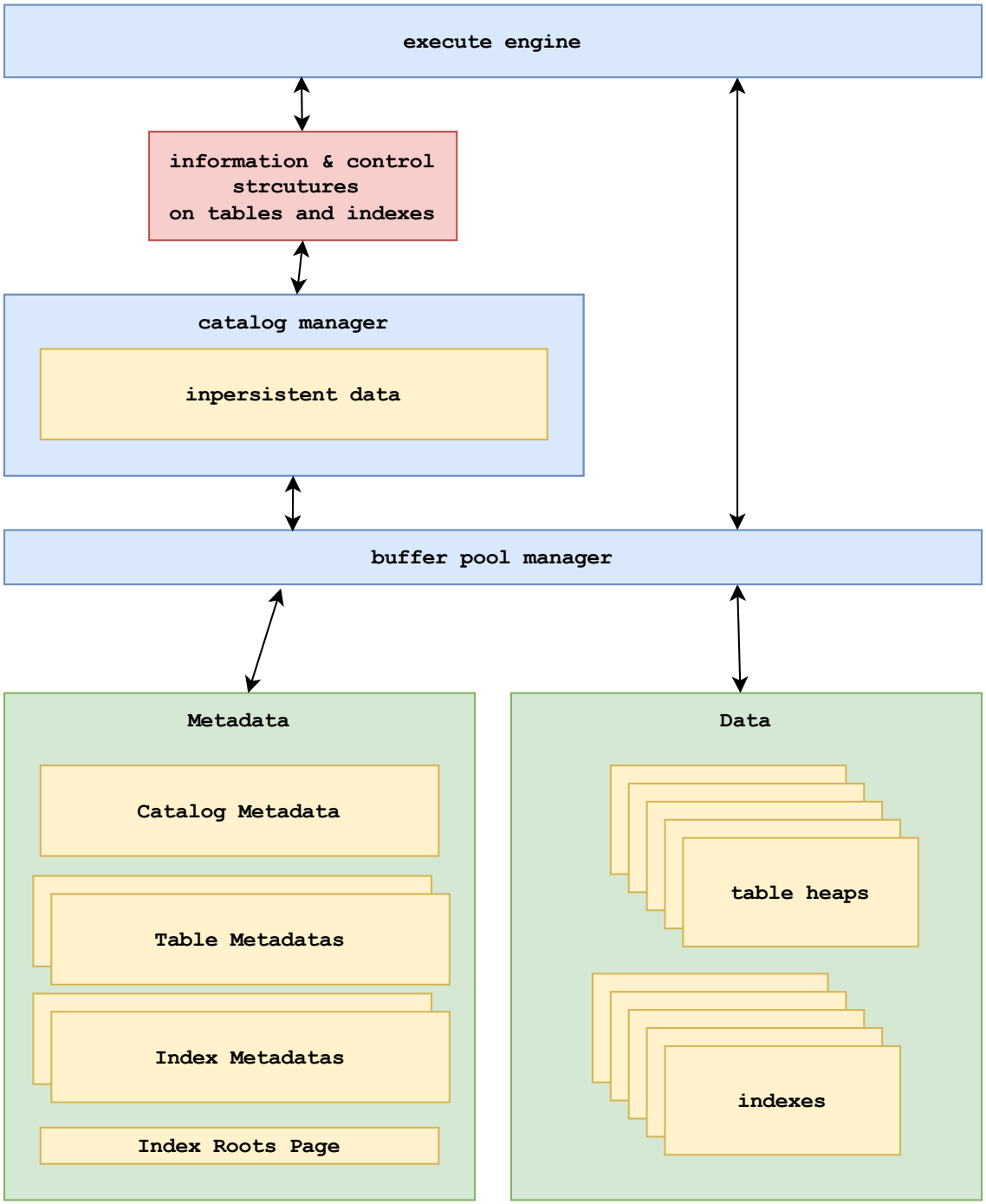
CatalogManager 管理了所有表、索引以及整个数据库的元信息，并把这些元信息与对应的数据结构关联起来，并为 Executor 提供查询、获取元信息与数据结构的接口。

具体的做法如下：ExecuteEngine 可以经由 CatalogManager 凭借 TableName , IndexName 获取表与索引的信息，也就是：IndexInfo 与 TableInfo。

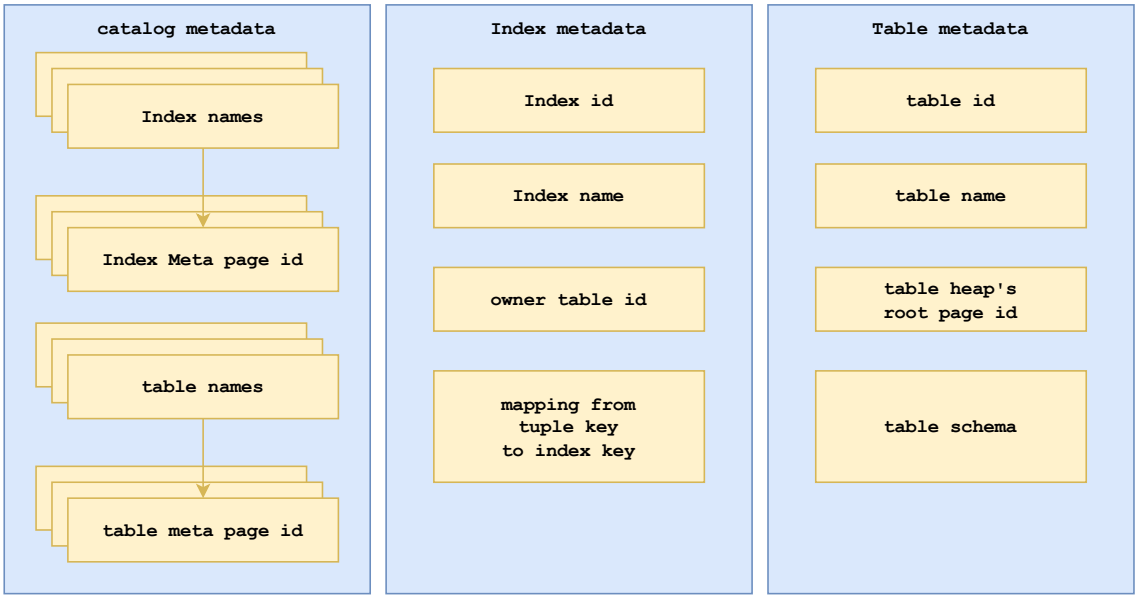
IndexInfo 中包含了 Index 的元信息：IndexMetaData，并且 IndexMetaData 直接存储在磁盘中。同理，TableInfo 包含了 Table 的元信息：TableMetaData。除此之外，IndexInfo 也包含了指向 BPlusTreeIndex 的指针，TableInfo 包含了指向 TableHeap 的指针。Executor 可以通过这些来判断一个索引或者表是否存在、列与索引的定义、以及操纵索引与表中实际存储的数据。

此外，CatalogManager 还管理了整个数据库中哪些表、哪些索引，以及索引的根节点信息。所有索引的根节点信息存储在一个单独的名为 `INDEX_ROOTS_PAGE` 的用户页中。

以上介绍的相互关系可以用下图表示。



CatalogMetaData 、IndexMetaData、TableMetaData 的结构如下图所示:



TableInfo 、IndexInfo 的数据成员如下：

```
1
2  class IndexInfo {
3   private:
4    IndexMetadata *index_meta_;
5    Index *index_;
6    TableInfo *table_info_;
7    IndexSchema *key_schema_;
8    MemHeap *heap_;
9  };
10
11 class TableInfo {
12  private:
13   TableMetadata *table_meta_;
14   TableHeap *table_heap_;
15   MemHeap *heap_; /** store all objects allocated in table_meta and table
         heap */
16 };
```

- CreateTable(const std::string &table_name, TableSchema *schema, Transaction *txn, TableInfo *&table_info) ： 创建表。
  1. 检查表是否重名，若重名，返回错误。
  2. 为表分配 root page 与 TableMetaPage
  3. 创建 TableMetaData 对象
  4. 创建 TableInfo 对象

5. 写入对应的页中
6. 更新 CatalogMeta 与 CatalogManager

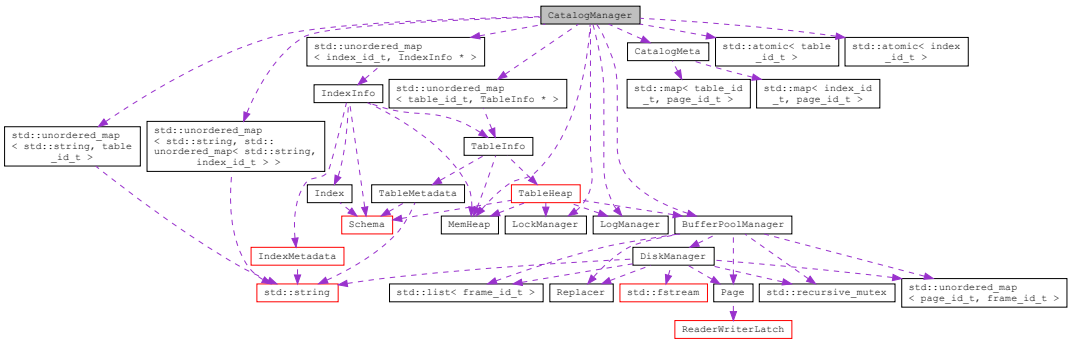
- GetTable(const std::string &table_name, TableInfo *&table_info);
  1. 检查表是否存在，若不存在，返回错误
  2. 从 TableInfo 的 map 中取出 TableInfo *
- GetTables(std::vector<TableInfo *> &tables) const;
  1. 遍历 TableInfo 的 map，逐个存入数组。
- `CreateIndex(const std::string &table_name, const std::string &index_name, const std::vector<std::string> &index_keys, Transaction *txn, IndexInfo *&index_info);`
  1. 检查表是否存在，若不存在，返回错误。
  2. 检查索引是否重名，若重名，返回错误。
  3. 检查列是否在表中存在，若不存在，返回错误
  4. 为索引分配 index root page 与 index meta page
  5. 创建 IndexMetadata 对象
  6. 创建 IndexInfo 对象
  7. 更新 CatalogMeta 与 CatalogManager
  8. 返回 IndexInfo * 对象的指针
- `GetIndex(const std::string &table_name, const std::string &index_name, IndexInfo *&index_info) const;`
- `GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes) const;`
  1. 查询 map<std::string, map<std::string, index_id_t»，返回对应的全部 IndexId
- `DropTable(const std::string &table_name);`
  1. 检查表是否存在，若不存在，返回错误。
  2. 调用 DropIndex，删除表上的全部索引
  3. 调用 TableHeap->FreeHeap，删除堆表中的全部数据。
  4. 释放 TableMetaPage。
  5. 析构 TableHeap、TableMetaData、TableInfo 对象。

6. 更新 CatalogManager 与 CatalogMeta

- `DropIndex(const std::string &table_name, const std::string &index_name);`

1. 检查索引是否存在，若不存在，返回错误。
2. 调用 Index->Destroy，删除索引中的全部数据。
3. 释放 IndexMetaPage。
4. 析构 BplusTreeIndex,IndexMetaData,IndexInfo 对象。
5. 更新 IndexRootsPage。
6. 更新 CatalogManager 与 CatalogMeta。

类关系图如下：



2.5 Replacer 的优化与多种 Replacer 的实现

2.5.1 LRUReplacer 的优化

在普遍的实现中，LRUReplacer 的算法如下：

- 对于每个 frame, 维护 lastUsedTime, 代表该 frame 上一次被访问的时间。
- 维护一个 list, 包含了可以被替换的 fid
- `UnPin(frame_id_t fid)`：

1. 将所有在 list 中的 fid 对应的 lastUsedTime 加 1
2. 将 fid 对应的加入 list，并设置 lastUsedTime 为 0
3. 时间复杂度：$O(N)$

- `Pin(frame_id_t fid)`：

1. 将 fid 从 list 中移除。

2. 时间复杂度：$O(1)$

- `Victim()`：

  1. 遍历所有 list 中的 fid
  2. 找到 lastUsedTime 最大的 fid
  3. 从 list 中移除 fid，并返回 fid
  4. 时间复杂度：$O(N)$

由于 `Unpin` 和 `Pin` 函数的使用极为频繁，因此 `Unpin` 的复杂度为 $O(N)$ 是不可接受的。因此，本项目放弃原框架中 `unordered_set` 实现，改用数组实现，并提供了以下等价的优化算法：

- 对于每个 frame, 维护 lastUsedTime, 代表该 frame 上一次被访问的时间。
- 维护一个 list, 包含了可以被替换的 fid
- 维护 min_time, 表示所有 lastUsedTime 中的最小值。
- `UnPin(frame_id_t fid)`：

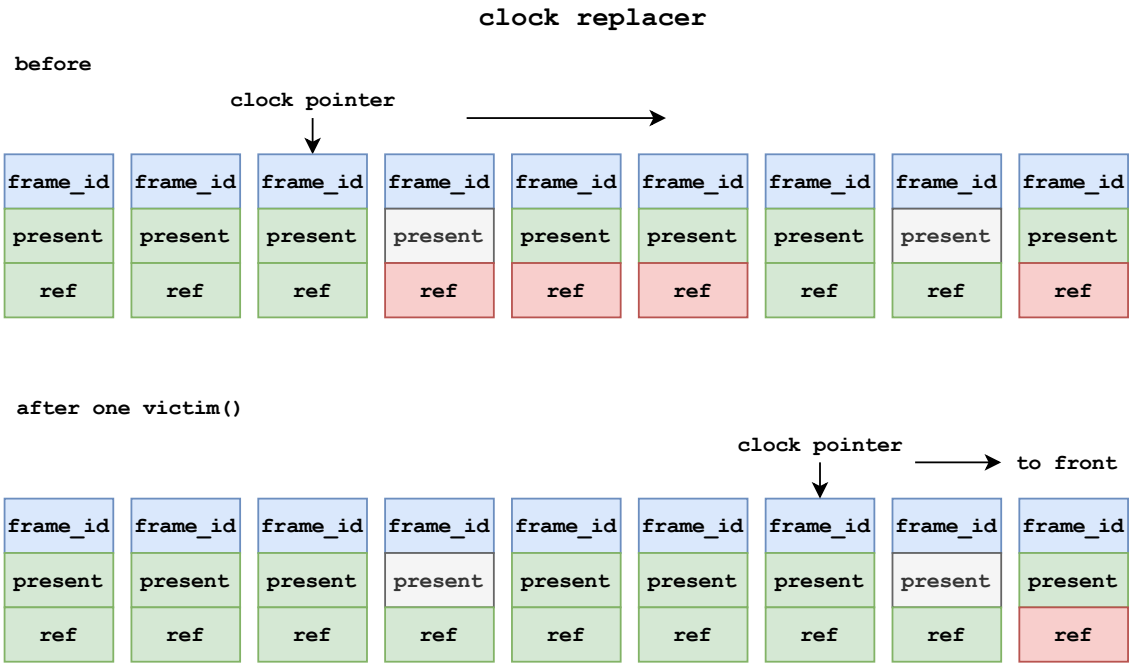
  1. 将 fid 对应的加入 list，并设置 lastUsedTime 为 min_time - 1
  2. 若发生溢出，则把所有的 lastUsedTime 加 1。
  3. 均摊时间复杂度：$O(1)$

- `Pin` 与 `Unpin` 原理与之前的大致相同。

2.5.2 ClockRepalcer 的实现

Clock Replacer, 顾名思义，像钟表一样的 replacer。Clock replacer 维护一个环形数组，其中的 Clock pointer 在每次 victim() 被调用时，循环地向后寻找第一个未被访问并且 present 的项，并返回该项。

简单原理图如下，ref 表示是否已被访问。每次 clock pointer 扫描到一项时，就将 ref 设置为 false。



后续的测试标明,clock replacer 的性能与 lru replacer（优化后）相当，都远大于优化之前的 lru replacer, 这是由于 clock replacer 的 unpin 、pin 操作都是 $O(1)$ 的。

具体实现如下:

- `Victim`:
  1. 从 clock pointer 向后遍历所有项，若达到结尾则返回开头。
  2. 寻找第一个 ref 为 false 且 present 的项，同时把经过的项的 ref 设置为 false。
- `Unpin`:
  1. 将该项对应的 ref 设为 true
  2. 将该项对应的 present 设为 true
- `Pin`:
  1. 将该项对应的 present 设为 false

2.6 Index 的重构和去模板化

在原项目框架中，`BPlusTreeIndex`、`BPlusTreeIndex` 为模板类，接受三个模板参数 `KeyType`，`ValueType`，`KeyComparator`，功能分别为:

- `KeyType`： 键的类型

- `ValueType`：值的类型
- `KeyComparator`：用于比较键的类

这种设计存在多种缺点。其中部分缺点是：

1. 浪费空间。`KeyType` 只支持特定长度的 key（在原框架中，这些长度分别是 4,8,16,32,64）。这意味着对于长度略大于 2 的次幂的那些 key，将会有接近一半的存储空间被浪费。
2. 灵活性差。key 具体需要多少长度，在运行期间是未知的，这就要求我们动态地根据 key 的长度对模板类进行实例化，破坏了封装性与代码的整洁。其次，模板类不可能实例化所有可能的长度，因此能够支持的 key 长度存在上限。
3. 代码膨胀。如果要对 Keysize 进行更加细粒度的划分，将不可避免的实例化大量的模板类，这将造成巨幅的代码膨胀。

当然，这种设计也存在一些优点，比如代码编写比较容易。

因此，本项目放弃了模板的设计，改用单一的 `BPlusTree` ，`BPlusTreeIndex` ，`BPlusTreePage` 类实现 B+ 树的操作，提高了灵活性，且节省了空间。具体做法是：

1. 将 keysize 作为成员保存在 `BPlusTreePage` 中。该成员在 `BPlusTreePage` 被创建时初始化。
2. 将 max_size 作为成员保存在 `BPlusPage` 中，代表该 Page 最多能保存的 Entry 数量。该成员在 `BPlusTreePage` 被创建时初始化。
3. 将 `BPlusTreeLeafPage` 的 `ValueType` 固定为 `RowId`
4. 将 `BPlusTreeInternalPage` 的 `ValueType` 固定为 `page_id_t`
5. 将 `KeyType` 固定为 `IndexKey`，`IndexKey` 的成员如下：

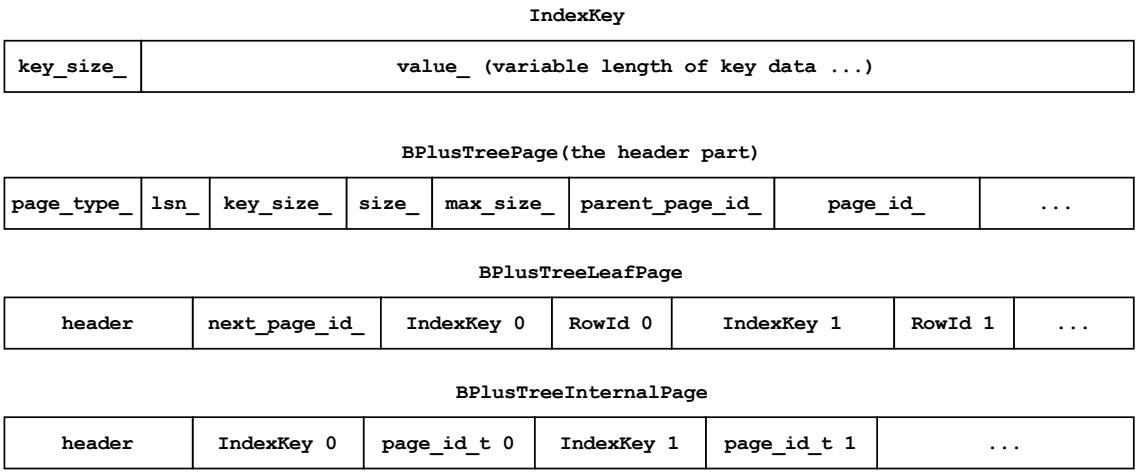
```
1 struct IndexKey{
2   uint8_t keysize;
3   char value[0]; // 柔性数组
4 }
```

6. GenericComparator 同时也去掉模板参数，变为 IndexKeyComparator。由于 IndexKey 中包含了 keysize 成员，key 的长度已知，因此可以进行比较。

`BPlusPage` 成员如下：

```
class BPlusTreePage{
  // ...
  private:
   IndexPageType page_type_;
   lsn_t lsn_;
   int key_size_; //键长度
   int size_; //当前键数
   int max_size_; //最大键数
   page_id_t parent_page_id_;
   page_id_t page_id_;
};
```

相关内存布局如下：



2.7 内存管理机制的多种优化

2.7.1 内存管理机制的优化

基于原框架中 MemHeap 的思想，本项目进一步优化了对内存的管理。

在本项目中，涉及到大量类对象的动态创建与销毁，内存管理是一个十分重要的方面。这是由于：

- 如果对动态创建的对象的生命周期管理不当，很容易造成内存泄露，尤其是在处理万级别的数据量时，使得程序无法持续运行。
- 大量临时对象的创建与销毁，若采用原始的内存分配方式，将导致极大的性能开销。
- 原有的 SimpleMemHeap 性能不足。

基于 MemHeap 的内存管理方式，我们提出以下两个概念：

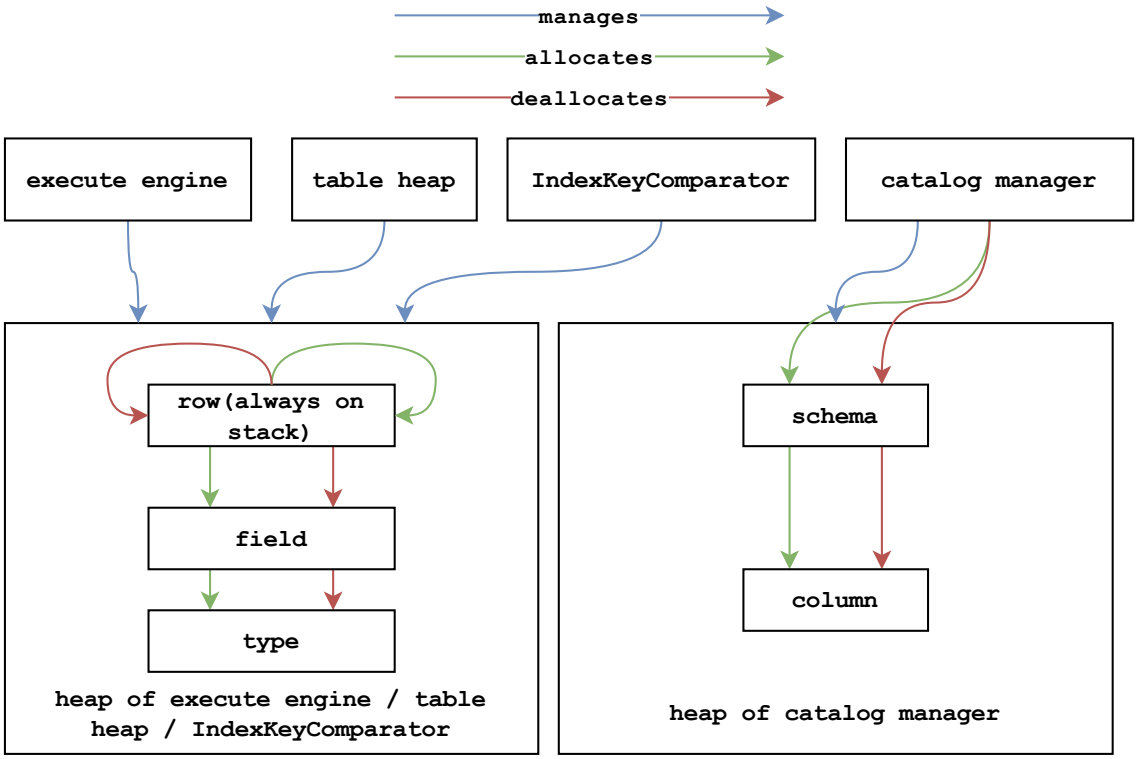
- “主对象”：成员包括 MemHeap，在 heap 上分配或者销毁自身的成员，且负责 MemHeap 的创建与销毁。
- “从对象”：成员包括 MemHeap，在 heap 上分配或者销毁自身以及自身的生源，但无权创建或者销毁 Memheap 的对象。
- 对于无需堆方式内存管理的对象，不在考虑范围之内。

之所以有这样的划分，是由于 memheap 的创建与销毁具有一定的性能开销。若 memheap 也由临时对象维护，那么频繁的创建与删除 memheap（比如原框架中 row 对象）将占用大量时间，且无法复用已分配的内存。因此，只有那些能够在内存中驻留较长时间的对象，才能对 memheap 进行管理。

基于以上概念，我们做出以下划分：

- 主对象：`ExecuteEngine`,`___Manager`,`TableHeap`,`IndexKeyComparator`
- 从对象：`Schema`,`Column`,`Row`,`Field`,`Type`

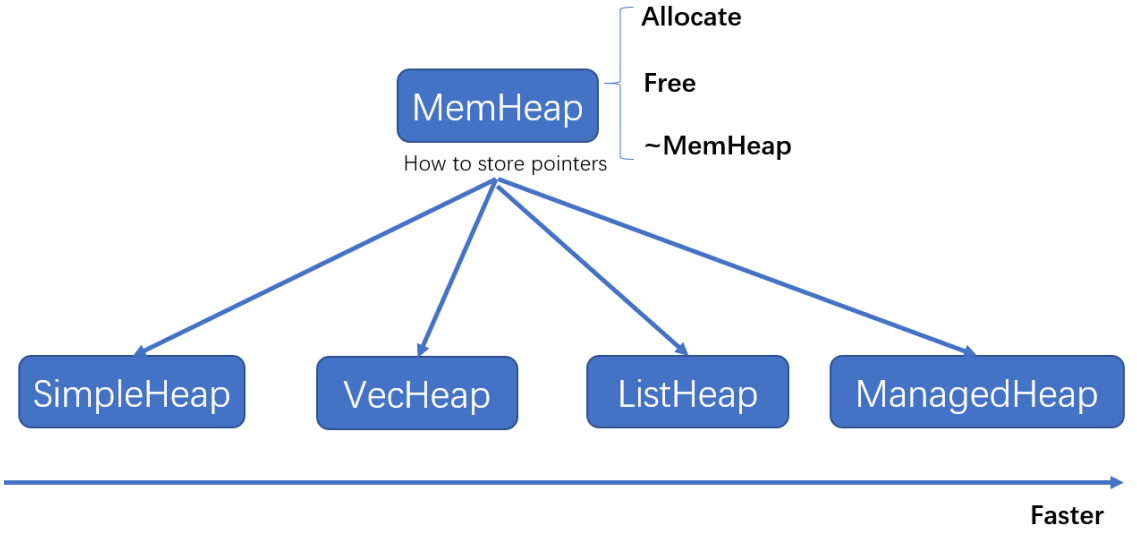
这样一来，便形成了如下的内存管理关系图：



最终，本项目实现了 0 内存泄露。（仅有的内存泄漏来自于 googletest，此外，还修复了 parser 模块中部分内存泄露。）

当然，每次在比较 key 的时候都对 Row 进行 deserialize，以及 row 对象的频繁创建与销毁，依然占据了除去文件读写以外 80% 的时间。在一个数据库系统之中，是否应该对于同一个 record 创建如此多的临时拷贝，并且通过反序列化的方式来进行比较，我认为是一个值得思考的问题。

2.7.2 高效内存池的实现



此外，本项目实现了一个高效快速的内存池：ManagedHeap。

MemHeap 用于更有效地管理内存分配和回收，析构会自动释放分配的内存空间，尽量避免内存泄露的问题，也能减少频繁 malloc 对性能的影响。

MemHeap 内部维护了已经分配空间的所有的指针集合，并在 free 时进行删除释放，析构时全部删除释放。在 MiniSQL 中主要为堆表、row、heap，catalog、executor 所需要生成的对象分配空间，其分配空间的操作（Allocate 函数）在程序执行的过程在被调用的频率极高（其中大部分是 row 的 heap 为每个 field 分配空间），在实际性能测试（perf 分析）的过程中发现其对性能的影响极大，占用了主要的时间（优化前，debug 模式插入带索引的 10 万行约 300 秒）

VecHeap：在分析中我们发现 SimpleHeap 内部通过维护一个 unordered_set 来存储指针，这种数据结构在查找时速度较快。但我们发现这样会导致插入的复杂度较高，而 free 往往是析构时一并执行的。相比于单个 free 操作，单个 allocate 的操作要频繁的多，所以我们将 unordered_set 改成了 vector，即 VecHeap，发现速度提升了（从 300 秒加快至 200 秒左右）

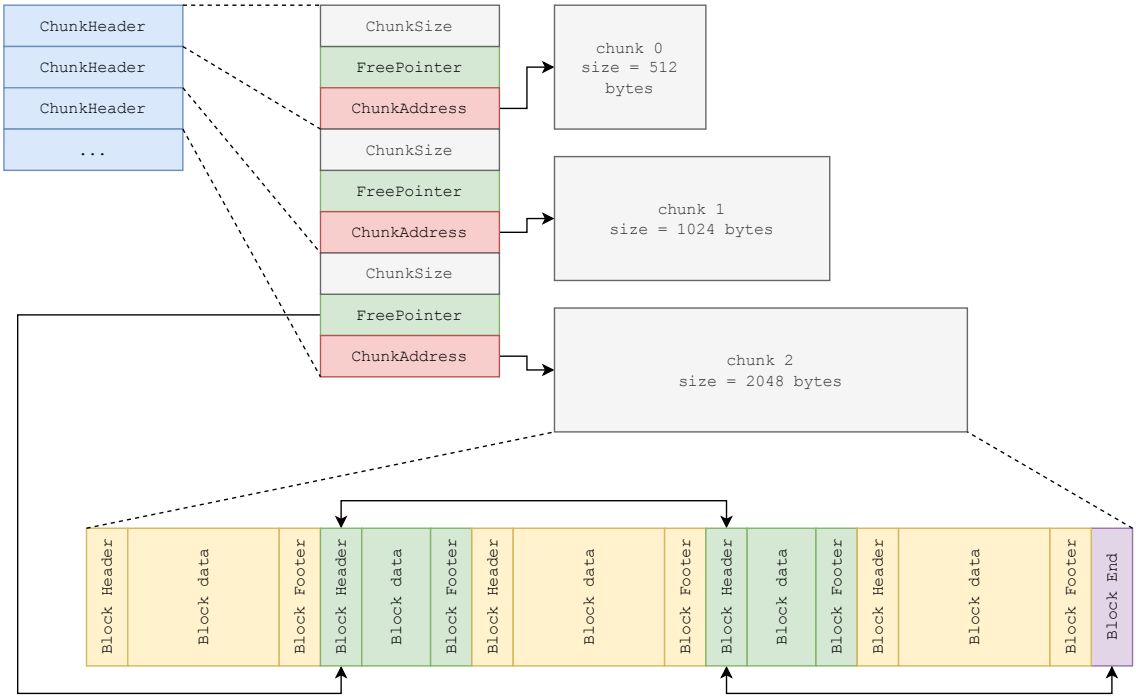
ListHeap：但是 VecHeap 的性能仍不是很理想，其 push_back 依然占据了程序执行的大部分时间（约 60% 80%），后来我们发现，由于在 MiniSQL 中大部分 Allocate 的场景是 row 的 MemHeap 去维护它的每个 field 的空间，而 field 的数量通常不会很大（个位数，或几十个），在这种较少的数据量下，使用 STL 提供的 vector 或 set 来维护可能显得臃肿而没必要。因此我们手写了简单的链表用来存储指针，使得插入复杂度为 O(1)，而删除则需要遍历。在这种轻量级的数据结构下，速度得到了大幅提升，debug 模式下插入带索引的 10 万行的时间加快到了约 34 秒

ManagedHeap：

设计思想

在数据库运行的过程中，会有大量临时的 Row 对象与 Field 对象不断被生成或者销毁。对于这些临时的对象，无需再重新调用 operaotr new –delete 或者 malloc - free 进行内存的申请和释放，这些 system call 本身的性能开销就比较大。只需要重复利用之前已经被申请过的内存即可。ManagedHeap 中先前被分配的内存在被释放时，该内存块不再被返还给操作系统，而是被 ManagedHeap 标记为 free，后续内存的分配可以直接复用这块内存，大大降低了内存申请与释放的开销。

总体架构



ManagedHeap 由若干个 Chunk 组成，每个 Chunk 代表一块较大的内存。相邻 Chunk 的大小以 2 为倍数倍增。在每个 Chunk 中，内存被线性地从前往后划分为若干个 Block，每个 Block 的状态为已分配或空闲。在每个 Block 中，首部和尾部各自维护了一个完全相同的 BlockHeader，记录了该 Block 的分配信息，其中包括：

- 该 Block 的大小
- 该 Block 是否已被分配
- 若该 Block 未被分配，记录上一个 free Block 相对 chunk 起始的偏移
- 若该 Block 未被分配，记录上一个 free Block 相对 chunk 起始的偏移

ChunkHeader 数组维护了每一个 Chunk 的信息，其中包括：

- Chunk 起始地址

- Chunk 大小，以字节计
- 该 Chunk 中第一个 free block

这样设计的优点在于，不仅能通过链表的结构快速的获取到空闲的块，还能根据某一个块的地址快速获取到相邻块的信息，从而完成块的分割或者合并。

ManagedHeap 的分配与释放实现如下：

- `Allocate(size)`：
  1. 计算能够容纳 size 大小的第一个 chunk。
  2. 从该 chunk 的第一个 free block 开始，沿着链表查找，若未找到则从下一个 chunk 中继续，直到找到满足大小的 free block。
  3. 若该 block 的大小显著大于 size，则对 block 进行分割，将新产生的空闲 block 加入 free list 中。
  4. 将该 block 从双向链表中删除，并对 chunk 的 free pointer、链表中前后 Block 头中的指针进行更新。
- `Free(void * p)`：
  1. p1 = p - sizeof(BlockHeader)，计算得控制块的地址。
  2. 检查魔数，判断该块内存是否由本 heap 分配。若否，抛出错误。
  3. 将该 chunk 的 free pointer 设置为 p1, 并把该块标记为 free。
  4. 将该块的相邻的也为 free 的 block 与自身合并，并更新相关的链表指针。

容易看到，`ManagedHeap` 无论是 `Allocate` 还是 `Free` 都只需要 $O(1)$ 的时间复杂度。并且在打开 release 优化之后，ManagedHeap 的运行速度得到了更大幅度的提升。采用 ManagedHeap 之后，插入带索引的 10 万行记录时间减少到了 10 秒左右。

3 个人总结

通过本项目，我了解并实现了数据库系统一种可能的具体架构，加深了对数据库系统底层架构的认识，也了解了在数据库程序运行过程中几个性能瓶颈所在。同时，本项目也极大地提升了我的工程能力与代码组织能力，能够着手编写较多文件的大型项目。最终项目能够正确运行，并支持海量数据的基本操作，非常有收获感。

不足的是，在整个项目完成之后，我才想到了一种通过优化索引存储的方式来极大地加速比较过程的方法。其次，ManagedHeap 的实现还存在很多缺点，比如：对于大量碎片化的内存，占用额外内存空间过多，时间复杂度常数过大，且对缓存不友好，后续的改进可以参考微软出品的 mi-malloc。此外，由于时间原因，未能实现事务管理、真正的查询优化，还无法支持普遍意义上的 SQL 语句。