**miniSQL个人模块设计报告**

郭永军 3200102126

1. Bitmap and diskmanager
   1. bitmap 位页图实现

这一模块实现了一个简单的位页图，一个位页图用于管理他所能记录的数据的使用与否的情况，如果是bit为1，则为已分配的数据页，如果bit位为0，则对应的数据页未分配。一般来说，一个数据页为4K，一个位页图同样是一个数据页，因此除去头部的metadata，后面的bit都可以用来记录数据页是否被分配。主要函数如下：

* 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）；
* BitmapPage::DeAllocatePage(page\_offset)：回收已经被分配的页；
* BitmapPage::IsPageFree(page\_offset)：判断给定的页是否是空闲（未分配）的。

Bitmap数据页的数据结构：|-- page\_allocated\_ --|-- next\_free\_page\_ --|....|

* 1. Diskmanager实现

这一模块在位页图的基础上实现了更大的一个diskmanager，实际上相当于一个更大的位页图，可以管理更多的数据页。通过一种类似套娃的方式，将前面所说的位页图及其管理的数据页视作一个整体，然后作为一个小块放在一个更大的metadata里面，这个metadata中的每一个数据都记录了一个前面所说的分区的信息。数据结构如下：

* Disk meta page：| Meta Page | Free Page BitMap 1 | Page 1 | Page 2 | ....  
   | Page N | Free Page BitMap 2 | Page N+1 | ... | Page 2N | ... |

相关函数：

* DiskManager::AllocatePage()：从磁盘中分配一个空闲页，并返回空闲页的逻辑页号；
* DiskManager::DeAllocatePage(logical\_page\_id)：释放磁盘中逻辑页号对应的物理页。
* DiskManager::IsPageFree(logical\_page\_id)：判断该逻辑页号对应的数据页是否空闲。
* 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。

1. Index manager
   1. B+树数据页
      1. BPlusTreePage

这是作为一个基类实现的数据结构，作为一个page，它包含一个B+树的节点所需要的基本信息，不管是叶子节点还是中间节点。以下为一个基础节点所应该包含的基础信息：

* 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。

其中page\_type\_是一个枚举类型的变量，用来指示这一个节点是中间节点还是中间节点。Lsn在本实验中未用到。Size\_指示当前节点已经包含了多少的key-value，即有多少个值，其中value是作为一个指针，对于中间节点来说，他指向下一层的节点，对于一个叶子节点来说，它包含对应key的pageid，也即相当于指向对应的Row。Parent\_page\_id同样相当于一个指向父亲节点的指针，page\_id则是当前节点所在的page。

本类包含以下函数：

* bool IsLeafPage() const //这一函数判断本page是否为叶子节点
* bool IsRootPage() const; //这一函数判断本page是否为根节点
* void SetPageType(IndexPageType page\_type); //根据参数设置本page为叶子节点还是中间节点
* int GetSize() const; //返回目前本节点容纳的key-value键值对的数目
* void SetSize(int size); //设置当前page所容纳的键值对数目为参数size
* void IncreaseSize(int amount); //增加当前page的size，增加数目为amount
* int GetMaxSize() const; //返回这个节点所能容纳的最大键值对的数目
* void SetMaxSize(int max\_size); //设置当前节点所能容纳的最大键值对数目
* int GetMinSize() const; //返回这个节点所能容纳的最小键值对的数目
* page\_id\_t GetParentPageId() const; //返回当前节点的父亲节点
* void SetParentPageId(page\_id\_t parent\_page\_id);//设当前节点的父亲节点为parent\_page\_id
* page\_id\_t GetPageId() const; //返回当前节点的page\_id
* void SetPageId(page\_id\_t page\_id); //设当前节点的page\_id\_为page\_id
  + 1. BPlusTreeInternalPage

这是一个由BPlusTreePage派生出来的类，相较于BPlusTreePage来说，它多包含了关于节点内部储存的键值对的信息，也即按照顺序储存的m个键和m+1个指针，这些指针实际上记录的是子节点的page\_id，对于一个中间节点来说，节点数量既不能超过他的最大值，也不能小于他规定的最小值。应当在必要的时候将这个节点分裂（超过最大值）或重新分配（小于最小值）,以下为本类中包含的相关函数：

* void Init(page\_id\_t page\_id, page\_id\_t parent\_id = INVALID\_PAGE\_ID, int max\_size = INTERNAL\_PAGE\_SIZE); //将本节点初始化，初始化相关的参数：page\_id, parent\_page\_id, max\_size
* KeyType KeyAt(int index) const; //返回位于第index位的key值
* void SetKeyAt(int index, const KeyType &key); //设置位于index位的key的值位key
* int ValueIndex(const ValueType &value) const; //返回value所在的index
* ValueType ValueAt(int index) const; //返回位于index位的value
* ValueType Lookup(const KeyType &key, const KeyComparator &comparator) const; //遍历节点，找到key所应该对应的value
* void PopulateNewRoot(const ValueType &old\_value, const KeyType &new\_key, const ValueType &new\_value); //create a new root page and insert these initial values
* int InsertNodeAfter(const ValueType &old\_value, const KeyType &new\_key, const ValueType &new\_value); //insert新的键值对new\_key, new\_value到old\_value之后
* void Remove(int index); //删除位于index处的键值对
* ValueType RemoveAndReturnOnlyChild(); //删除本节点中的最后一个键值对，然后返回最后一个value
* void MoveAllTo(BPlusTreeInternalPage \*recipient, const KeyType &middle\_key, BufferPoolManager \*buffer\_pool\_manager); //将本届点中的所有键值对都移动到recipient节点中去，然后通过middle\_key保持中间节点一致性，并更新父亲节点中的相关的键值对
* Void MoveHalfTo(BPlusTreeInternalPage \*recipient, BufferPoolManager \*buffer\_pool\_manager); //将当前节点中的一般的节点移动到相邻节点中去，并更新父亲节点相关的键值对
* void MoveFirstToEndOf(BPlusTreeInternalPage \*recipient, const KeyType &middle\_key, BufferPoolManager \*buffer\_pool\_manager);//将当前节点中的第一个节点移动 到recipient的最后面去，并调整当前节点的键值对，设置被移动的子节点的父亲节点
* void MoveLastToFrontOf(BPlusTreeInternalPage \*recipient, const KeyType &middle\_key, BufferPoolManager \*buffer\_pool\_manager); //将当前节点的最后一个节点移动到recipient的最前面，调整recipent的键值对使其能够容纳键值对，调整被移动节点的parent\_id
* void CopyNFrom(MappingType \*items, int size, BufferPoolManager \*buffer\_pool\_manager); //复制size个键值对到当前目录里面
* void CopyLastFrom(const MappingType &pair, BufferPoolManager \*buffer\_pool\_manager); //复制一个键值对到末尾
* void CopyFirstFrom(const MappingType &pair, BufferPoolManager \*buffer\_pool\_manager);

//复制一个键值对到最前面

* + 1. BPlusTreeLeafPage

这一个类同样是由BPlusTreePage派生而来，其中包含m个键以及m个值，其中键由一个或者多个field序列化得到，值位对应的RowID。本类的数据结构相较于基类多了一个信息next\_page\_id（在迭代器的时候用到）以及相应的键值对。下面为相关函数：(由于leafPage和internalPage的很多函数类似，在此不过多赘述)

void Init(page\_id\_t page\_id, page\_id\_t parent\_id = INVALID\_PAGE\_ID, int max\_size = LEAF\_PAGE\_SIZE);  
 page\_id\_t GetNextPageId() const;  
 void SetNextPageId(page\_id\_t next\_page\_id);  
 KeyType KeyAt(int index) const;  
 int KeyIndex(const KeyType &key, const KeyComparator &comparator) const;  
 const MappingType &GetItem(int index);  
 int Insert(const KeyType &key, const ValueType &value, const KeyComparator &comparator);  
 bool Lookup(const KeyType &key, ValueType &value, const KeyComparator &comparator) const;  
 int RemoveAndDeleteRecord(const KeyType &key, const KeyComparator &comparator);  
 // Split and Merge utility methods  
 void MoveHalfTo(BPlusTreeLeafPage \*recipient, BufferPoolManager \*buffer\_pool\_manager);  
 void MoveAllTo(BPlusTreeLeafPage \*recipient, const KeyType &middle\_key, BufferPoolManager \*buffer\_pool\_manager);  
 void MoveFirstToEndOf(BPlusTreeLeafPage \*recipient, const KeyType &middle\_key,BufferPoolManager \*buffer\_pool\_manager);  
 void MoveLastToFrontOf(BPlusTreeLeafPage \*recipient, const KeyType &middle\_key,BufferPoolManager \*buffer\_pool\_manager);  
private:  
 void CopyNFrom(MappingType \*items, int size);  
 void CopyLastFrom(const MappingType &item);  
 void CopyFirstFrom(const MappingType &item);

* 1. B+树索引

在实现了基础的节点及其操作之后，需要进行完成整个B+树的实现及操作，主要操作包括创建，插入，删除，查找和释放。B\_Plus\_tree的数据结构如下：

* index\_id\_: 本B+树对应的索引的ID
* root\_page\_id：根节点的page\_id
* buffer\_pool\_manager：用于管理分配和回收相关的数据页
* Comparator： 用于比较键值的大小
* leaf\_max\_size： 叶子结点的最大键值对数量
* internal\_max\_size\_：中间节点的最大键值对数量

相关的函数：

public:  
 explicit BPlusTree(index\_id\_t index\_id, BufferPoolManager \*buffer\_pool\_manager, const KeyComparator &comparator, int leaf\_max\_size = LEAF\_PAGE\_SIZE, int internal\_max\_size = INTERNAL\_PAGE\_SIZE);  
 //创建B+树并初始化  
 bool IsEmpty() const;  
 //Returns true if this B+ tree has no keys and values  
 bool Insert(const KeyType &key, const ValueType &value, Transaction \*transaction = nullptr);  
 //Insert a key-value pair into this B+ tree.  
 void Remove(const KeyType &key, Transaction \*transaction = nullptr);  
 //Remove a key and its value from this B+ tree.  
 bool GetValue(const KeyType &key, std::vector<ValueType> &result, Transaction \*transaction = nullptr);  
 //return the value associated with a given key  
 INDEXITERATOR\_TYPE Begin();  
 //返回一个指向初始键值对的迭代器  
 INDEXITERATOR\_TYPE Begin(const KeyType &key);  
 //返回指向特定key的迭代器  
 INDEXITERATOR\_TYPE End();  
 //返回指向结尾的键值的迭代器  
 Page \*FindLeafPage(const KeyType &key, bool leftMost = false);  
 //找到key所在的叶子节点，若leftMost为true，则返回最小key所在的叶子节点  
 bool Check();  
 // used to check whether all pages are unpinned  
 void Destroy();  
 // destroy the b plus tree  
 void PrintTree(std::ofstream &out);

//以特定格式打印整个B+树  
private:

//-------------------用于插入的相关函数----------------//  
 void StartNewTree(const KeyType &key, const ValueType &value);  
 //当插入第一个键值对的时候新建整个树  
 bool InsertIntoLeaf(const KeyType &key, const ValueType &value, Transaction \*transaction = nullptr);  
 //插入一个键值对到leafPage。首先寻找到对应的leafPage，然后插入，如果大于最大值，则需要Split  
 void InsertIntoParent(BPlusTreePage \*old\_node, const KeyType &key, BPlusTreePage \*new\_node, Transaction \*transaction = nullptr);  
 //如果由于Split产生了一个新的节点，则需要将这个新的节点插入到对应的父亲节点中去，同时如果父亲节点的键值对数量超过了最大值，则需要对父节点递归调用split以及InsertIntoParent  
 template<typename N>  
 N \*Split(N \*node);  
 //将当前节点分裂，并将一半的键值对移动到新的节点中去

//-----------------用于删除一个键值对---------------------//  
 template<typename N>  
 bool CoalesceOrRedistribute(N \*node, Transaction \*transaction = nullptr);  
 //删除一个节点之后，如果该节点的键值对数量小于该节点的最小数量，则需要调用本函数，本函数首先根据相邻的节点数量判断调用Coalesce还是Redistribute  
 template<typename N>  
 bool Coalesce(N \*\*neighbor\_node, N \*\*node, BPlusTreeInternalPage<KeyType, page\_id\_t, KeyComparator> \*\*parent, int index, Transaction \*transaction = nullptr);  
 //当节点与相邻节点的键值对的和小于节点所能容纳的最大键值对时，调用本函数，将右边的节点的所有键值对移动到左边的节点中去，然后删除本节点，调整相关孩子节点以及父亲节点的相关信息。在删除节点之后，父亲节点进行判断，如果键值对小于最小值，则对父亲节点调用CoalesceOrRedistribute。  
 template<typename N>  
 void Redistribute(N \*neighbor\_node, N \*node, int index);  
 //当节点与相邻节点的键值对数量之和大于键值对所能容纳的最大键值对数量，调用本函数，将相邻节点中的一个键值对移动到本节点中，调整相关的子节点以及父节点的信息。  
 bool AdjustRoot(BPlusTreePage \*node);  
 //当调整节点的过程中（主要为删除节点），如果root节点的最后一个键值被删除了，组该节点需要被删除，然后将他的子节点作为新的根节点  
 void UpdateRootPageId(int insert\_record = 0);

//在root\_page中更新现在的根节点

* 1. B+树索引迭代器

这是为了遍历B+树而专门设计的迭代器。数据结构为：

int index = 0;  
B\_PLUS\_TREE\_LEAF\_PAGE\_TYPE\* leaf = nullptr;  
BufferPoolManager \*buffer\_pool\_manager = nullptr;

* index：为在对应的leafpage中现在的迭代器指向第几个key
* Leaf： 为现在指向的key所在的leafpage
* Buffer\_pool\_manager；主要用于fetchpage

相关的函数:

explicit IndexIterator();  
IndexIterator(B\_PLUS\_TREE\_LEAF\_PAGE\_TYPE\* leaf\_, int index\_, BufferPoolManager\* buffer\_pool\_manager\_);  
~IndexIterator();  
/\*\* Return the key/value pair this iterator is currently pointing at. \*/  
const MappingType &operator\*();  
  
/\*\* Move to the next key/value pair.\*/  
IndexIterator &operator++();  
/\*\* Return whether two iterators are equal \*/  
bool operator==(const IndexIterator &itr) const;  
/\*\* Return whether two iterators are not equal. \*/  
bool operator!=(const IndexIterator &itr) const;

模块总结：index manager这一部分主要用于进行索引相关的B+树的构建，插入，删除，查找等操作，并对外提供接口，其中用到模板类等相关的知识，不仅对于更加深刻理解C++有更多的帮助，同时对于索引这一部分概念的相关问题也有了更加清晰的认识。