CATALOG MANAGER实验报告

姓名： 张鑫

学号：3200102809

专业：计算机科学与技术

0 绪论

本模块由我一人完成。Catalog Manager 负责管理和维护数据库的所有模式信息，包括：

- 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。
- 表中每个字段的定义信息，包括字段类型、是否唯一等。
- 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。

1 序列化

1.1 Catalog Meta

Catalog Meta存储着数据库系统的元数据，诸如，表id到表meta页，索引id到索引meta页的映射。

Catalog Meta的成员如下：

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

meta data中表和索引的映射是需要进行持久化存储的。

因而对meta data的序列化次序为： MAGIC_NUM -> size of (table_meta_pages) -> size of(index_meta_pages) -> 遍历table_meta_pages序列化 -> 遍历index_meta_pages_序列化。

1.2 Table Meta

Table Meta中存储的是对应着一张表的持久化数据，包括表的id，表的名字，表对应的堆表的第一页。

Table Meta的私有成员如下：

```
    table_id_t table_id_;
    std::string table_name_;
    page_id_t root_page_id_;
    Schema *schema_;
```

其中schema的序列化直接调用schema底层的序列化接口即可。

1.3 Index Meta

Index Meta的序列化和Table Meta类似。但vector类型的key_map_和string类型的name都遵循序列化长度->序列化值的模式。

1.4 Table Info

Table Info是表信息在内存中的存在形式，成员如下：

```
    TableMetadata *table_meta_;
    TableHeap *table_heap_;
    MemHeap *heap_; /** store all objects allocated in table_meta and
  table heap */
```

其中table_heap_是使用table_meta_中的first page id创建的堆表。

而table heap是该表在创建时，Catalog Manager为其分配的首页id创建的table heap对象。

MemHeap为Catalog Manager传参

2 表和索引的恢复

Catalog Manager中维护的是数据库系统的Meta信息，当数据库引擎启动时，Catalog Manager需要将存在于磁盘中的Meta信息读取出，在内存中重构该数据库的模式对象。

而在Catalog中，我们维护的Meta信息分别是，数据库中表id以及其对应的元数据页，索引id以及其对应的元数据页。

因此，当Catalog Manager进行恢复时，我们需要将Catalog Meta Page Id页的数据读出，建立id到元数据页的映射，再利用这个映射，读取对应元数据页，依次建立表name到表id的映射，表name到表信息对象的映射，表name到表上索引的id-name哈希表映射，以及索引id到索引对象的映射。

具体操作如下

```
    // fetch catalog meta data
```

```
  char* buf = this->buffer_pool_manager_-
>FetchPage(CATALOG_META_PAGE_ID)->GetData();
  this->catalog_meta_ = CatalogMeta::DeserializeFrom(buf,this->heap_);
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID,false);
  // need not to fetch index root page


  // create table info
  // fetch map from catelog manager

  for(auto &it:this->catalog_meta_->table_meta_pages_){
    buf = this->buffer_pool_manager_->FetchPage(it.second)->GetData();
    this->buffer_pool_manager_->UnpinPage(it.second,false);
    TableMetadata* table_meta
;TableMetadata::DeserializeFrom(buf,table_meta,this->heap_);
    if(it.first>=next_table_id_) next_index_id_  = it.first+1;
    table_names_.insert(std::pair<std::string,table_id_t>(table_meta-
>GetTableName(),it.first));
    TableInfo* table_info = TableInfo::Create(this->heap_);
    TableHeap* table_heap = TableHeap::Create(this-
>buffer_pool_manager_,table_meta->GetFirstPageId(),table_meta-
>GetSchema(),this->log_manager_,this->lock_manager_,this->heap_);
    table_info->Init(table_meta,table_heap);
    this->tables_.insert(std::pair<table_id_t,TableInfo*>
(it.first,table_info));
    this->index_names_.insert(std::pair<std::string,
std::unordered_map<std::string, index_id_t>>(table_meta-
>GetTableName(),std::unordered_map<std::string,index_id_t>()));
  }

  for(auto &it:this->catalog_meta_->index_meta_pages_){
    buf = this->buffer_pool_manager_->FetchPage(it.second)->GetData();
    this->buffer_pool_manager_->UnpinPage(it.second,false);
    IndexMetadata* index_meta;
    IndexMetadata::DeserializeFrom(buf,index_meta,this->heap_);
    if(next_index_id_<=it.first) next_index_id_ = it.first + 1;

    table_id_t table_id = index_meta->GetTableId();
    TableInfo* table_info = tables_.find(table_id)->second;
    IndexInfo* index_info = IndexInfo::Create(this->heap_);
    std::string table_name = table_info->GetTableName();
    auto& table_index_map = this->index_names_.find(table_name)->second;
    table_index_map.insert(std::pair<std::string,index_id_t>(index_meta-
>GetIndexName(),it.first));
    index_info->Init(index_meta,table_info,this->buffer_pool_manager_);
    this->indexes_.insert(std::pair<index_id_t,IndexInfo*>
(it.first,index_info));
  }
```

3 表和索引的创建和删除

3.1 创建/删除表

创建表时除了更新内存中的Catalog Manager中的映射关系之外还应该为其分配元数据页，并更新Catalog Meta Page中的id到元数据页的映射关系。另外还应当为该表创立独立的堆表对象，为堆表对象分配首页。

与之对应的，删除表时要释放表信息和堆表占用的数据页，更新映射关系。另外由于堆表可能占用很多页，那么删除表时就需要迭代地释放堆表页。

3.2 创建/删除索引

和创建删除表类似地需要更新Catalog中的映射关系，以及Catalog Meta Page中持久化的元信息。

但不同的是表建立需要额外分配堆表，而索引建立和删除则是要额外更新Index Root Page中的元信息。

(IndexRootPage中存储的是索引ID到索引B+树的根所在页)。

另外，创建索引需要调用堆表迭代器，将堆表中所有记录同步到B+树中，删除时则需要遍历删除B+树。

注：根据OOP设计思想，涉及到B+树的操作部分全部由底层实现，Catalog层无感知，只需要确定B+树的键的类型。