姓名：沈韵泓
学号：3200104392
班级：软工2001
其余小组成员：梁可愉、赵伊蕾、黄亦霄
[注] 个人报告内容为Record Manager模块的Record相关部分与Catalog Manager模块

MiniSQL个人报告

1 Record Manger

1.1 模块概述

在MiniSQ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。

在Record Manager中，所有的记录以堆表（Table Heap）的形式进行组织。堆表由多个数据页构成链表，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，表中所有的记录都是无序的。在对记录进行查找操作时，返回符合条件记录的起始迭代器，具体的迭代访问工作由 Executor 负责。

1.2 主要功能

- 记录（Record）的序列化：

  将Row/Column/Schema对象中的信息转化为字节流（`char*`）写入给定的数据页中（通过`buf`指针定位），将逻辑数据转换为存储在文件中的物理数据。成功后返回在该过程中`buf`指针向前推进的字节数。

- 记录（Record）的反序列化：

  将磁盘数据页中以字节流(`char*`)形式存储的物理数据恢复为内存中的逻辑数据。成功后在传入的参数中返回恢复的Row/Column/Schema对象，函数本身返回在反序列化过程中`buf`指针向前推动的字节数。

1.3 对外提供的接口

```
1. uint32_t Row::SerializeTo(char *buf, Schema *schema)
   //将该Row对象序列化至从buf指向位置开始的数据页地址中，返回该过程中buf推进的字节数
2. uint32_t Row::DeserializeFrom(char *buf, Schema *schema)
   //从buf指向位置开始的数据页地址中恢复信息至当前Row对象中，返回该过程中buf推进的字节数
3. uint32_t Row::GetSerializedSize(Schema *schema) const
   //返回该Row对象序列化过程中buf推进的字节数
4. uint32_t Column::SerializeTo(char *buf)
   //将该Column对象序列化至从buf指向位置开始的数据页地址中，返回该过程中buf推进的字节数
5. uint32_t Column::DeserializeFrom(char *buf, Column *&column, MemHeap *heap)
   //在传入的MemHeap中新建Column对象，并从buf指向位置开始的数据页地址中恢复信息至该Cloumn对象中，在传入参数的Column中返回，函数本身返回该过程中buf推进的字节数
6. uint32_t Column::GetSerializedSize() const
   //返回该Column对象序列化过程中buf推进的字节数
7. uint32_t Schema::SerializeTo(char *buf) const
   //将该Schema对象序列化至从buf指向位置开始的数据页地址中，返回该过程中buf推进的字节数
8. uint32_t Schema::DeserializeFrom(char *buf, Schema *&schema, MemHeap *heap)
   //在传入的MemHeap中新建Schema对象，并从buf指向位置开始的数据页地址中恢复信息至该Schema对象中，在传入参数的Schema中返回，函数本身返回该过程中buf推进的字节数
```

9. uint32_t Schema::GetSerializedSize() const
   //返回该Schema对象序列化过程中buf推进的字节数

1.4 设计思路

- Row

  Row对象的序列化/反序列化实现使用了NullBitmap进行辅助记录。

  在序列化的开始，先写入MAGIC_NUM，随后写入field的总数与NullBitMap记录空字段，再依次调用Field的序列化函数对Row对象进行序列化。序列化存储格式如下：

```
 Row format:
-----------------------------------------
| Field Nums | Null bitmap | Field-1 | ... | Field-N |
-----------------------------------------
```

  在反序列化的开始，读入MAGIC_NUM验证对象类型，随后读入NullBitMap，根据NullBitmap对对应的字段进行反序列化或标为NULL后跳过。

- Column

  序列化时先写入MAGIC_NUM，随后依次写入`name_`，`type_`，`len_`，`table_ind_`，`nullable`，`unique_`等属性。

  反序列化时，先读入MAGIC_NUM验证对象类型，随后依次读入上述属性完成对Column对象的重载。

- Schema

  序列化时先写入MAGIC_NUM，再序列化该schema中包含的column总数，随后依次调用column的序列化函数进行序列化。

  反序列化时先读入MAGIC_NUM验证对象类型，再读入n_column，进行相应次数的Column::DeserializeFrom函数的调用。

1.5 整体架构

在该模块中，Row的相关操作调用了Field类中的相关函数，Schema的相关操作调用了Column类中的相关函数。Row与Schema均对外提供接口，该模块总体架构如下：

1.6 关键函数和代码

- Row
  - 序列化

    在Row的序列化过程中通过Null Bitmap对空字段进行标注，并调用了field的序列化函数进行实现。

```
//Serialize sequence: page_id -> slot_num_ -> n_fields -> null bitmap
-> n*(TypeId, (lenstr), value)
uint32_t Row::SerializeTo(char *buf, Schema *schema) const {
  char *cur = buf;
  uint32_t s_size = 12;                          //Serialized Size
  //Serialize rid_ : page_id_ -> slot_num_
  MACH_WRITE_INT32(cur, this->rid_.GetPageId());   cur += 4;
  MACH_WRITE_UINT32(cur, this->rid_.GetSlotNum()); cur += 4;
```

```
  //Serialize n_fields
  int n_fields = (int)this->GetFieldCount();
  MACH_WRITE_INT32(cur, n_fields);                    cur += 4;
  //Serialize null_bitmap
  int size = n_fields/8 + 1;                          //size of null
bitmap
  char *bitmap = new char[size];                      //null_bitmap: 0-
Not NULL, 1-NULL
  memset(bitmap, 0x0, size*sizeof(char));             //initialize null
bitmap
  for(int i = 0; i < n_fields; i++)
  {
    if(this->fields_[i]->IsNull())
    {//mark '1' at the i-th bit
      int addr = i/8;
      int ofs  = i%8;
      unsigned char tmp = 0x1 << ofs;
      bitmap[addr] |= tmp;
    }
  }
  for(int i = 0; i < size; i++)
  {
    MACH_WRITE_TO(char, cur, bitmap[i]); cur += 1;
  }
  s_size += size;
  //Serialize fields_value;
  for(int i = 0; i < n_fields; i++)
  {
    uint32_t ofs = this->fields_[i]->SerializeTo(cur);
    cur += ofs;
    s_size += ofs;
  }
  return s_size;;
}
```

- 反序列化

首先通过MAGIC_NUM核实对象类型，随后读入Null BItmap对非NULL的field进行序列化操作。

```
//Deserialize Sequence: page_id -> slot_num_ -> null bitmap -> n*Field
uint32_t Row::DeserializeFrom(char *buf, Schema *schema) {
  char *cur = buf;
  uint32_t d_size = 12;                    //Deserialized Size
  //Deserialized RowId
  page_id_t p_id = MACH_READ_INT32(cur);  cur += 4;
  uint32_t s_num = MACH_READ_UINT32(cur); cur += 4;
  RowId r_id(p_id, s_num);
  SetRowId(r_id);
```

```
  //Read null bitmap
  int n_fields = MACH_READ_INT32(cur);    cur += 4;
  int size = n_fields/8 + 1;
  d_size += size;
  char *bitmap = new char[size];
  for(int i = 0; i < size; i++)
  {
    bitmap[i] = MACH_READ_FROM(char, cur);
    cur += 1;
  }
  //Deserialize Fields
  for(int i = 0; i < n_fields; i++)
  {
    int addr = i/8;
    int ofs = i%8;
    unsigned char tmp = 0x1 << ofs;
    TypeId t_id = schema->GetColumn((uint32_t)i)->GetType();
    Field *f = nullptr;
    uint32_t move = 0;
    if((bitmap[addr] & tmp) > 0)  move = f->DeserializeFrom(cur, t_id,
&f, true , heap_); //field[i] is null
    else                          move = f->DeserializeFrom(cur, t_id,
&f, false, heap_); //field[i] is not null
    cur += move;
    d_size += move;
    fields_.push_back(f);
  }
  return d_size;
}
```

- Get Serialized Size

```
uint32_t Row::GetSerializedSize(Schema *schema) const {
  uint32_t res = 12;
  int n_fields = (int)this->fields_.size();
  res += n_fields/8 + 1;
  for(int i = 0; i < n_fields; i++)
  {
    res += this->fields_[i]->GetSerializedSize();
  }
  return res;
}
```

- Column
  - 序列化

    首先序列化MAGIC_NUM，随后依次对Column对象的各成员变量进行序列化

```
uint32_t Column::SerializeTo(char *buf) const {
  char *cur = buf;
  MACH_WRITE_TO(int32_t, cur, COLUMN_MAGIC_NUM);      cur += 4;
  MACH_WRITE_TO(bool, cur, IsNullable());             cur += 1;
  MACH_WRITE_TO(int32_t, cur, GetName().size());      cur += 4;//l_name
  MACH_WRITE_STRING(cur, GetName()) ;                 cur +=
GetName().size();
  MACH_WRITE_TO(TypeId, cur, GetType());              cur += 4;
  MACH_WRITE_TO(uint32_t, cur, GetLength());          cur += 4;
  MACH_WRITE_TO(uint32_t, cur, GetTableInd());        cur += 4;
  MACH_WRITE_TO(bool, cur, this->unique_);            cur += 1 ;
  return GetName().size()+4*5+2;
}
```

- 反序列化

首先读入MAGIC_NUM验证对象类型，随后按照序列化的顺序读取各属性，完成反序列化

```
uint32_t Column::DeserializeFrom(char *buf, Column *&column, MemHeap
*heap) {
    char *cur = buf;
  //Deserialize MAGIC_NUM & ASSERT
  uint32_t magic_num = MACH_READ_UINT32(cur); cur += 4;
  ASSERT(magic_num == COLUMN_METADATA_MAGIC_NUM, "Unexpected Type!");
  //Deserialize all attributes
  bool isNull = MACH_READ_FROM(bool, cur);            cur += 1;
  int len_name = MACH_READ_FROM(int32_t, cur);        cur += 4;
  std::string c_name;
  for(int i = 0; i < len_name; i++)
  {
    c_name += MACH_READ_FROM(char, cur);
    cur += 1;
  }
  TypeId t_id = MACH_READ_FROM(TypeId, cur);          cur += 4;
  uint32_t c_len = MACH_READ_FROM(uint32_t, cur);     cur += 4;
  uint32_t t_ind = MACH_READ_FROM(uint32_t, cur);     cur += 4;
  bool isUnique = MACH_READ_FROM(bool, cur);          cur += 1;

  if (t_id == kTypeChar)
    column = ALLOC_P(heap, Column)(c_name, t_id, c_len, t_ind, isNull,
isUnique);
  else
    column = ALLOC_P(heap, Column)(c_name, t_id, t_ind, isNull,
isUnique);
  return len_name + 4*5 + 2;
}
```

- Get Serialized Size

```
uint32_t Column::GetSerializedSize() const {
return GetName().size() + 4*5 + 2;
}
```

- Schema
  - 序列化

    调用了Column的序列化函数进行实现。

```
//Serialize sequence: MAGIC_NUM -> n_Column -> n*()
uint32_t Schema::SerializeTo(char *buf) const {
  char *cur = buf;
  uint32_t s_size = 8;                              //Serialized Size
  //Serialize MAGIC_NUM
  MACH_WRITE_UINT32(cur, this->SCHEMA_MAGIC_NUM); cur += 4;
  //Serialize <vector>columns_
  uint32_t n_column = this->GetColumnCount();
  MACH_WRITE_UINT32(cur, n_column); cur += 4;
  uint32_t i = 0;
  while(i < n_column)
  {
    uint32_t ofs =  GetColumn(i)->SerializeTo(cur);
    s_size += ofs;
    cur += ofs;
    i++;
  }

  return s_size;
}
```

  - 反序列化

    首先通过MAGIC_NUM验证对象类型，再通过调用column的反序列化函数完成实现。

```
uint32_t Schema::DeserializeFrom(char *buf, Schema *&schema, MemHeap
*heap) {
  if (schema != nullptr) {
    LOG(WARNING) << "Pointer to schema is not null in schema
deserialize." << std::endl;
  }
  uint32_t d_size = 8;
  char *cur = buf;
  //Deserialize MAGIC_NUM & ASSERT
  uint32_t magic_num = MACH_READ_UINT32(cur); cur += 4;
  ASSERT(magic_num == SCHEMA_METADATA_MAGIC_NUM, "Unexpected Type!");
  //Deserialize Columns
```

```
    int n_column = MACH_READ_UINT32(cur); cur += 4;
    std::vector<Column *> columns(n_column);
    for(int i = 0; i < n_column; i++)
    {
      uint32_t ofs = columns[i]->DeserializeFrom(cur, columns[i], heap);
      cur += ofs;
      d_size += ofs;
    }
    schema = new Schema(columns);
    return d_size;
  }
```

- Get Serialized Size

  调用Column的相关函数。

```
  //4*(MAGIC_NUM + n_column) + ∑ c_serialize_size
  uint32_t Schema::GetSerializedSize() const {
    uint32_t res = 8;
    uint32_t n_column = this->GetColumnCount();
    for(uint32_t i = 0; i < n_column; i++)
    {
      res += GetColumn(i)->GetSerializedSize();
    }
    return res;
  }
```

2 Catalog Manager

2.1 模块概述

在MiniSQL中，Catalog Manager 负责管理和维护数据库中所有表（及其字段定义）和索引的所有定义信息，并在这些信息被创建、修改、删除后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。

2.2 主要功能

- 目录元信息的序列化

  将CatalogMeta/TableMetadata/IndexMetadata对象中的信息转化为字节流（`char*`）写入给定的数据页中（通过`buf`指针定位），将逻辑数据转换为存储在文件中的物理数据。成功后返回在该过程中`buf`指针向前推进的字节数。

- 目录元信息的反序列化

  将磁盘数据页中以字节流(`char*`)形式存储的物理数据恢复为内存中的逻辑数据。成功后在传入的参数中返回恢复的CatalogMeta/TableMetadata/IndexMetadata对象，函数本身返回在反序列化过程中`buf`指针向前推动的字节数。

- 表与索引的管理

  - CatalogManager的构造：

在首次建立数据库实例时(`init=true`)初始化元数据，在后续重新打开该实例时(`init=false`)从数据库文件中加载所有的表和索引信息，置于内存中。

- 表与索引的建立：

  根据Executor提供的信息新建TableInfo/IndexInfo对象，并更新CatalogManager中的相应信息，实时将改变后的信息落盘。若建立失败，则返回相应的错误信息（`DB_TABLE_ALREADY_EXIST`，`DB_TABLE_NOT_EXIST`，`DB_INDEX_ALREADY_EXIST`，`DB_COLUMN_NAME_NOT_EXIST`）。

- 表与索引的删除：

  根据Executor提供的信息删除CatalogManager中的相应信息，删除表时级联删除建立在该表上的所有索引，实时将改变后的信息落盘。

- 表与索引的查找：

  根据Executor提供的信息在对应的数据库中对表/索引进行查找，若查找失败则返回对应的错误信息（`DB_TABLE_NOT_EXIST`，`DB_INDEX_NOT_FOUND`），成功则返回对应的TableInfo/IndexInfo（GetTables/GetIndexes时返回由该类对象组成的vector容器）。

2.3 对外提供的接口

```
1. CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager,
LockManager *lock_manager,LogManager *log_manager, bool init)
   //init = true 时，初始化CtalogMeta中的数据，将Table/Index数标为0并存盘
   //init = false时，从数据库文件中加载所有的表与索引信息置于内存中
2. dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema
*schema,Transaction *txn, TableInfo *&table_info)
   //检测是否已存在同名Table，若存在，返回DB_TABLE_ALREADY_EXIST;
   //若不存在，根据输入的TableSchema创建新的TableInfo在参数中进行返回，更新CatalogMeta中的
相关数据并落盘，最终返回DB_SUCCESS
3. dberr_t CatalogManager::DropTable(const string &table_name)
   //检测是否存在给定表，若不存在，返回DB_TABLE_NOT_EXIST;
   //若存在，删除该Table上的所有Index后删除该Table的相关信息，更新CatalogMeta中的相关数据并
落盘，最终返回DB_SUCCESS
4. dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables) const
   //在参数中返回该数据库下的所有TableInfo，函数返回DB_SUCCESS
5. dberr_t CatalogManager::GetTable(const table_id_t table_id, TableInfo
*&table_info)
   //根据给定的table_id检查是否存在对应表，若不存在，返回DB_TABLE_NOT_EXIST;
   //若存在，则在参数中返回要求的TableInfo，函数返回DB_SUCCESS
6. dberr_t CatalogManager::GetTable(const string &table_name, TableInfo
*&table_info)
  //根据给定的table_name检查是否存在对应表，若不存在，返回DB_TABLE_NOT_EXIST;
   //若存在，则在参数中返回要求的TableInfo，函数返回DB_SUCCESS
7. dberr_t CatalogManager::CreateIndex(const std::string &table_name, const
string &index_name, const std::vector<std::string> &index_keys, Transaction *txn,
IndexInfo *&index_info)
  //根据table_name检测是否存在给定Table，若不存在，返回DB_TABLE_NOT_EXIST;
  //检测该Table下是否已存在同名Index，若存在，返回DB_INDEX_ALREADY_EXIST;
```

```
  //若指定Table下不存在同名Index，根据输入的TableSchema创建新的IndexInfo在参数中进行返回，更新CatalogMeta中的相关数据并落盘，最终返回DB_SUCCESS
8. dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name)
   //检测是否存在给定索引，若不存在，返回DB_INDEX_NOT_EXIST;
   //若存在，删除该Index的相关信息，更新CatalogMeta中的相关数据并落盘，最终返回DB_SUCCESS
9. dberr_t CatalogManager::GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes) const
  //根据table_name检测是否存在给定Table，若不存在，返回DB_TABLE_NOT_EXIST;
  //若存在，在参数中返回该Table下的所有IndexInfo，函数返回DB_SUCCESS
10. dberr_t CatalogManager::GetIndex(const std::string &table_name, const std::string &index_name, IndexInfo *&index_info) const
  //根据table_name检测是否存在给定Table，若不存在，返回DB_TABLE_NOT_EXIST;
  //检测该Table下是否存在给定Index，若不存在，返回DB_INDEX_NOT_EXIST;
  //若存在，则在参数中返回要求的IndexInfo，函数返回DB_SUCCESS
```

2.4 设计思路

- 元信息的序列化

  按照MAGIC_NUM至其余成员变量的顺序进行序列化，新增指针`char* cur`指向当前插入位置，每次序列化信息后向前推进相应的字节数，最终返回指针`cur`与起始地址`buf`的距离差。

  序列化字符串时，需要先序列化该字符串的长度，再逐字符对string进行序列化。

- 元信息的反序列化

  在开始读入MAGIC_NUM，通过ASSERT语句确定反序列化的对象与预期相符。新增指针`char* cur`指向当前读取位置，每次反序列化信息后向前推进相应的字节数，最终返回指针`cur`与起始地址`buf`的距离差。

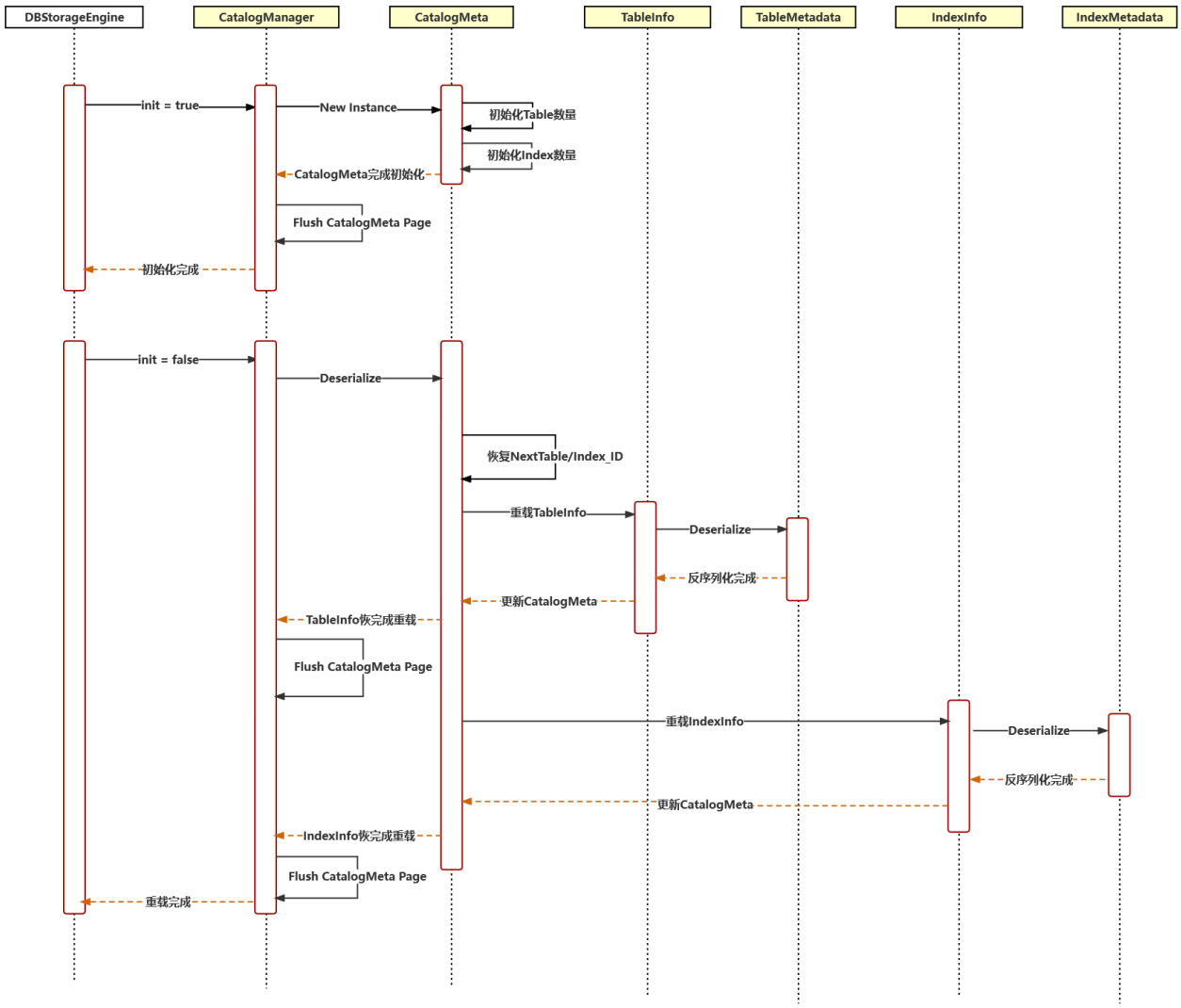
- 对表/索引的操作
  - 查询类操作仅对CatalogManager与CatalogMeta做读操作
  - 创建/删除类操作首先检查操作的合法性，随后由CatalogManager对相应的TableInfo/IndexInfo进行操作，将更改落盘。最后更新CatalogManager中的相关信息，并将更新后的数据落盘。

2.5 整体架构

整个模块由CatalogManager类提供对外接口，CatalogMeta、TableInfo、IndexInfo均通过CatalogManager的heap分配空间，并在CatalogManager析构时自动释放。TableMetadata与IndexMetadata的空间则分别由TableInfo和IndexInfo分配。

各子模块的整体架构如下：

以CatalogManager的构造函数为例，各子模块间的交互逻辑如下：



2.6 关键函数和代码

2.6.1 CatalogMeta

- 序列化

  使用指针`char* cur`指向当前待写入位置，随序列化过程移动。

  首先对MAGIC_NUM进行序列化，随后按顺序序列化成员变量，字符串变量需先序列化子不传长度，随后逐字符序列化整个字符串。

```
//Every data read/written in CatalogMeta::Serilization/Deserialization is of type-uint32_t, thus the buf move 4 Bytes every time.
//Serialize Sequence: MAGIC_NUM -> size_of_table_map -> table_map -> size_of_index_map -> index_map
void CatalogMeta::SerializeTo(char *buf) const {
  char *cur = buf;
  //Serialize MAGIC_NUM
  MACH_WRITE_UINT32(cur, this->CATALOG_METADATA_MAGIC_NUM); cur += 4;
  //Serialize Map - table_meta_pages_
  MACH_WRITE_UINT32(cur, (uint32_t)table_meta_pages_.size()); cur += 4;
  auto t_it = table_meta_pages_.begin();
  while(t_it != table_meta_pages_.end())
```

```
    {
      MACH_WRITE_UINT32(cur, t_it->first);
      MACH_WRITE_INT32(cur+4, t_it->second);
      cur += 8;
      t_it++;
    }
    //Serialize Map - index_meta_pages_
    MACH_WRITE_UINT32(cur, (uint32_t)index_meta_pages_.size()); cur += 4;
    auto i_it = index_meta_pages_.begin();
    while(i_it != index_meta_pages_.end())
    {
      MACH_WRITE_UINT32(cur, i_it->first);
      MACH_WRITE_INT32(cur+4, i_it->second);
      cur += 8;
      i_it++;
    }
  }
```

- 反序列化

  首先用断言验证待序列化对象的MAGIC_NUM符合预期，随后按照序列化顺序依次反序列化成员变量。

```
  CatalogMeta *CatalogMeta::DeserializeFrom(char *buf, MemHeap *heap) {
    CatalogMeta* catalog_meta = CatalogMeta::NewInstance(heap);
    char * cur = buf;
    //Deserialize MAGIC_NUM & ASSERT
    uint32_t magic_num = MACH_READ_UINT32(cur); cur += 4;
    ASSERT(magic_num == CATALOG_METADATA_MAGIC_NUM, "Unexpected Type!");
    //Deserialize Map - table_meta_pages_
    int size_t = (int)MACH_READ_UINT32(cur); cur += 4;
    for(int i = 0; i < size_t; i++)
    {
      uint32_t t_id = MACH_READ_UINT32(cur);
      uint32_t p_id = MACH_READ_INT32(cur+4);
      cur += 8;
      catalog_meta->table_meta_pages_.emplace(t_id, p_id);
    }

    //Deserialize Map - index_meta_pages_
    int size_i = (int)MACH_READ_UINT32(cur); cur += 4;
    for(int i = 0; i < size_i; i++)
    {
      uint32_t i_id = MACH_READ_UINT32(cur);
      uint32_t p_id = MACH_READ_INT32(cur+4);
      cur += 8;
      catalog_meta->index_meta_pages_.emplace(i_id, p_id);
    }
```

```
  return catalog_meta;
}
```

- Get Serialized Size

  根据实际序列化的成员变量类型及数量进行计算，每序列化一个字符串时，需要加上序列化其长度的代价`sizeof(int)`。

```
uint32_t IndexMetadata::GetSerializedSize() const {
  uint32_t l_name = index_name_.size();
  uint32_t l_map = key_map_.size();
  return l_name + 4*l_map + 20;
}
```

2.6.2 TableMetadata

其序列化、反序列化、GetSerializedSize原理与CatalogMeta一致

2.6.3 IndexMetadata

其序列化、反序列化、GetSerializedSize原理与CatalogMeta一致

2.6.4 Catalog Manager

- 构造函数：

  需要区分`init=true`与`init=false`的情况，分别进行相应处理，视情况分别进行初始化/重载操作。

```
CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager,
LockManager *lock_manager,
                               LogManager *log_manager, bool init)
        : buffer_pool_manager_(buffer_pool_manager),
lock_manager_(lock_manager),
          log_manager_(log_manager), heap_(new SimpleMemHeap()) {

  if(init)
  { //首次建立DBStorageEngine -> 初始化CatalogMeta
    cout << "initializing database..." << endl;
    this->catalog_meta_ = CatalogMeta::NewInstance(this->heap_);
    this->catalog_meta_->table_meta_pages_[next_table_id_] = -1;
    this->catalog_meta_->index_meta_pages_[next_index_id_] = -1;
    this->FlushCatalogMetaPage();
    // this->buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, false);
  }
  else
  {
    // 1. 从数据库文件中反序列化CatalogMeta的信息
    Page* c_meta_p = this->buffer_pool_manager_-
>FetchPage(CATALOG_META_PAGE_ID);
    this->catalog_meta_= CatalogMeta::DeserializeFrom(c_meta_p->GetData(),
this->heap_);
```

```
    this->buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, false);

    this->next_index_id_ = this->catalog_meta_->GetNextIndexId();
    this->next_table_id_ = this->catalog_meta_->GetNextTableId();
    if(this->catalog_meta_->table_meta_pages_.size())
      cout << "[Reload DB] There are: " << this->catalog_meta_-
>table_meta_pages_.size()-1 << " tables & ";
    else
      cout << "[Reload DB] There are: 0 tables & ";
    if(this->catalog_meta_->index_meta_pages_.size())
      cout << this->catalog_meta_->index_meta_pages_.size()-1 << "
indexs\n";
    else
      cout << "0 indexs\n";
    // 2. 根据CatlogMeta加载所有的表和索引信息，构建TableInfo和IndexInfo信息置于内
存中
    // this->LoadTable(const table_id_t table_id, const page_id_t page_id)
    // 2.1 加载所有的Table
    auto t_it = this->catalog_meta_->table_meta_pages_.begin();
    while(t_it != this->catalog_meta_->table_meta_pages_.end())
    {
      if(t_it->second == -1)
      {
        t_it++;
        continue;
      }
      //新建 table_info 并完成初始化
      TableInfo* table_info = TableInfo::Create(this->heap_);
      Page* p_t_meta = this->buffer_pool_manager_->FetchPage(t_it->second);
      TableMetadata* t_meta;
      TableMetadata::DeserializeFrom(p_t_meta->GetData(), t_meta,
table_info->GetMemHeap());
      TableHeap *t_heap = TableHeap::Create(buffer_pool_manager_, t_meta-
>GetFirstPageId(), t_meta->GetSchema(), log_manager_, lock_manager_,
table_info->GetMemHeap());
      table_info->Init(t_meta, t_heap);
      //更新CatalogManager中的相关信息
      this->table_names_.insert(pair<std::string, table_id_t>(t_meta-
>GetTableName(), t_meta->GetTableId()));
      this->tables_.insert(pair<table_id_t, TableInfo *>(t_meta-
>GetTableId(), table_info));
      this->index_names_.insert({t_meta->GetTableName(),
std::unordered_map<std::string, index_id_t>()});
      cout << "            TABLE:: " << t_meta->GetTableName() << endl;
      t_it ++;
    }
    // 2.2 加载所有的Index
```

```
    auto i_it = this->catalog_meta_->index_meta_pages_.begin();
    while(i_it !=this->catalog_meta_->index_meta_pages_.end())
    {
      if(i_it->second == -1)
      {
        i_it++;
        continue;
      }
      //新建 index_info 并完成初始化
      IndexInfo* index_info = IndexInfo::Create(this->heap_);
      Page* p_i_meta = this->buffer_pool_manager_->FetchPage(i_it->second);
      IndexMetadata* i_meta;
      IndexMetadata::DeserializeFrom(p_i_meta->GetData(), i_meta,
index_info->GetMemHeap());
      //初始化，似乎每次打开都要重新弄建立index
      //index不存盘的话，buffer_pool_manager有啥用?
      TableInfo* cur_t_info = this->tables_.find(i_meta->GetTableId())-
>second;
      index_info->Init(i_meta, cur_t_info, this->buffer_pool_manager_);
      //更新CatalogManager中的相关信息
      this->index_names_[cur_t_info->GetTableName()][index_info-
>GetIndexName()] = i_meta->GetIndexId();
      this->indexes_[i_meta->GetIndexId()] = index_info;
      cout << "           INDEX OF TABLE:: " << cur_t_info->GetTableName()
<< ", NAME:: " << index_info->GetIndexName() << endl;
      i_it ++;
    }
  }
}
```

- 刷新CatalogMetaPage：

  该页逻辑页码CATALOG_META_PAGE_ID默认为0。

  由于不存在清空数据页内容的函数，DELETE后重新分配的页码可能与要求不一致，故此处直接调用CatalogMeta的序列化函数，对数据页进行覆写。代码实现如下：

```
dberr_t CatalogManager::FlushCatalogMetaPage() const {
  // 直接拿序列化覆盖旧的信息
  // 1. 调用CatalogMeta序列化最新信息
  Page* c_meta_p = this->buffer_pool_manager_-
>FetchPage(CATALOG_META_PAGE_ID);
  this->catalog_meta_->SerializeTo(c_meta_p->GetData());
  // 2. 存盘
  this->buffer_pool_manager_->FlushPage(CATALOG_META_PAGE_ID);
  this->buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
  return DB_SUCCESS;
}
```

- 新建表与新建索引：
  - 新建表时需检查是否存在同名表
  - 新建索引时需检查是否存在对应表、是否存在同名索引、指定的Key是否存在

    不符合条件时，应返回对应的错误信息，此处以Create Index的实现为例：

```
dberr_t CatalogManager::CreateIndex(const std::string &table_name, const
string &index_name,
                                    const std::vector<std::string>
&index_keys, Transaction *txn,
                                    IndexInfo *&index_info) {
  //1.1 检测DB中是否已存在给定表
  auto c1 = this->table_names_.find(table_name);
  if(c1 == this->table_names_.end())
    return DB_TABLE_NOT_EXIST;
  // 存在->从CatalogManger中获取TableInfo
  TableInfo *t_info = tables_.find(c1->second)->second;
  //1.2 表中是否存在同名索引
  vector<IndexInfo *> t_indexes;
  GetTableIndexes(t_info->GetTableName(), t_indexes);//获取该表下的index list
  int k_size = t_indexes.size();
  for(int i = 0; i < k_size; i++)
  {
    if(t_indexes[i]->GetIndexName() == index_name)
    {
      return DB_INDEX_ALREADY_EXIST;
    }
  }
  //1.3 表中是否存在对应的Column，构建key_map
  std::vector<uint32_t> key_map;
  int n_key = int(index_keys.size());
  for(int j = 0; j < n_key; j++)
  {
    auto i = index_keys[j];
    uint32_t tkey;
    dberr_t error = t_info->GetSchema()->GetColumnIndex(i, tkey);
    if(error) return error;//返回Column不存在的错误提示
    key_map.push_back(tkey);
  }
  //3. 新建IndexInfo
  index_info = IndexInfo::Create(this->heap_);
  index_id_t i_id = this->next_index_id_;
  this->next_index_id_++;
  this->catalog_meta_->index_meta_pages_[next_index_id_] = -1;
  IndexMetadata *i_meta = IndexMetadata::Create(i_id, index_name, t_info-
>GetTableId(), k_map, index_info->GetMemHeap());
  // 序列化indexmeta，存盘
```

```
    page_id_t i_meta_p_id;
    this->buffer_pool_manager_->NewPage(i_meta_p_id);
    Page* i_meta_p = this->buffer_pool_manager_->FetchPage(i_meta_p_id);
    i_meta->SerializeTo(i_meta_p->GetData());
    this->buffer_pool_manager_->FlushPage(i_meta_p_id);
    this->buffer_pool_manager_->UnpinPage(i_meta_p_id, true);
    // 完成index_info::Init
    index_info->Init(i_meta, t_info, this->buffer_pool_manager_);
    //4. 更新CatalogManager中的相关信息
    this->index_names_[table_name][index_name] = i_id;
    this->indexes_.insert(pair<index_id_t, IndexInfo *>(i_id, index_info));
    //5. 更新CatalogMeta中的相应数据
    this->catalog_meta_->index_meta_pages_[i_id] = i_meta_p_id;
    //6. 将更新数据落盘
    this->FlushCatalogMetaPage();
    this->buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
    return DB_SUCCESS;
}
```

- 查询表与查询索引
  - 查询表：根据给定的table_name/table_id在对应的map中进行查找，不存在则返回DB_TABLE_NOT_EXIST
  - 查询索引：先根据给定的table_name确定是否存在给定表，再通过index_name在对应map中进行查找

    此处以Get Index的实现为例：

```
dberr_t CatalogManager::GetIndex(const std::string &table_name, const
std::string &index_name,
                                 IndexInfo *&index_info) const {
  //1. table_name -> index_name
  auto it_tb = this->index_names_.find(table_name);
  if(it_tb == this->index_names_.end())
    return DB_TABLE_NOT_EXIST;
  //2. index_name -> index_id
  auto it_idx = it_tb->second.find(index_name);
  if(it_idx == it_tb->second.end())
    return DB_INDEX_NOT_FOUND;
  //3. index_id -> index_info*
  index_info = this->indexes_.find(it_idx->second)->second;
  return DB_SUCCESS;
}
```

  - 查询所有表/索引：访问对应的map，并将所有的TableInfo/IndexInfo在vector中返回

    此处以Get Indexes的实现为例：

```
dberr_t CatalogManager::GetTableIndexes(const std::string &table_name,
std::vector<IndexInfo *> &indexes) const {
  //返回指定table下的所有Index_Info*
  //1. table_name -> <index_name, index_id>
  auto it_tb = this->index_names_.find(table_name);
  if(it_tb == this->index_names_.end())
    return DB_TABLE_NOT_EXIST;

  //2. 遍历 <index_name, index_id>, 实现 index_id -> index_info*
  auto m_idxs = it_tb->second;//<index_name, index_id>
  for(auto it_idx = m_idxs.begin(); it_idx != m_idxs.end(); it_idx++)
  {
    index_id_t cur_id = it_idx->second;
    indexes.push_back(indexes_.find(cur_id)->second);
  }
  return DB_SUCCESS;
}
```

- 删除表/删除索引
  - 删除索引：首先检查对应的表及索引是否存在，均存在后删除对应的数据页，并更新Catalog中的信息
  - 删除表：首先检查对应的表是否存在，若存在则先删除该表上的所有索引，再删除表对应的数据页并更新Catalog中的信息

    此处以Drop Table的实现为例：

```
dberr_t CatalogManager::DropTable(const string &table_name) {
  // 1. 检查是否存在该table
  auto c1 = this->table_names_.find(table_name);
  if(c1 == this->table_names_.end())
    return DB_TABLE_NOT_EXIST;
  // 1++ 删除该table下所有的index
  TableInfo* t_info = this->tables_.find(c1->second)->second;
  vector<IndexInfo *> t_indexes;
  GetTableIndexes(t_info->GetTableName(), t_indexes);
  int n_idx = t_indexes.size();
  for(int i = 0 ; i < n_idx; i++)
  {
    DropIndex(t_info->GetTableName(), t_indexes[i]->GetIndexName());
  }
  this->index_names_.erase(t_info->GetTableName());
  // 2. 根据t_meat中的信息，先将所有的table_page删除
  page_id_t cur_p_id = t_info->GetRootPageId();
  TablePage* cur_p = static_cast<TablePage *>(this-
>buffer_pool_manager_->FetchPage(cur_p_id));
  page_id_t next_p_id = cur_p->GetNextPageId();
  this->buffer_pool_manager_->DeletePage(cur_p_id);
```

```
        while(next_p_id != INVALID_PAGE_ID)
        {
          cur_p = static_cast<TablePage *>(this->buffer_pool_manager_-
      >FetchPage(next_p_id));
          cur_p_id = next_p_id;
          next_p_id = cur_p->GetNextPageId();
          this->buffer_pool_manager_->DeletePage(cur_p_id);
        }
        // 3. 根据CatalogMeta 中的信息，删除table_meta
        auto it = this->catalog_meta_->table_meta_pages_.find(c1->second);
        this->buffer_pool_manager_->DeletePage(it->second);
        // 4. 删除CalogMeta 中的相关信息，存盘
        this->catalog_meta_->table_meta_pages_.erase(c1->second);
        this->FlushCatalogMetaPage();
        // 5. 删除CatalogManager 中的相关信息
        this->tables_.erase(c1->second);
        this->table_names_.erase(table_name);
        // 不需要手动从内存中释放 t_info，会报double free的错误
        return DB_SUCCESS;
      }
```

2.6.5 IndexInfo

- 初始化

  IndexInfo的初始化主要分为三步：

  1. 初始化index_metadata和table_info
  2. 初始化key_map -> 通过shallowcopy实现，与TableSchema中的列共享同一份存储
  3. 调用CreateIndex以新建Index

```
void Init(IndexMetadata *meta_data, TableInfo *table_info, BufferPoolManager
*buffer_pool_manager) {
    // Step1: init index metadata and table info
    this->meta_data_ = meta_data;
    this->table_info_ = table_info;
    // Step2: mapping index key to key schema
    key_schema_ = Schema::ShallowCopySchema(this->table_info_->GetSchema(),
this->meta_data_->GetKeyMapping(), this->heap_);
    // Step3: call CreateIndex to create the index
    this->index_ = Index::CreateIndex(buffer_pool_manager_);
  }
```

其中，Index::CreateIndex应根据key的总长度确定B+树索引应该应用的模板类型，实现如下：

```
Index *CreateIndex(BufferPoolManager *buffer_pool_manager) {
    //1. 先计算索引key的最大长度
    vector<Column *> idx_columns;
    idx_columns = key_schema_->GetColumns();
```

```
    int n_key = idx_columns.size();
    int l_sum = 0;
    for(int i = 0; i < n_key; i++)
    {
      l_sum += idx_columns[i]->GetLength();
    }
    //2. 选择对应长度的索引模板
    if(l_sum <= 4)
    {
      void *mem = heap_->Allocate(sizeof(BPlusTreeIndex<GenericKey<4>,
RowId, GenericComparator<4>>));
      return static_cast<Index*>(new(mem)BPlusTreeIndex<GenericKey<4>,
RowId, GenericComparator<4>>(this->meta_data_->GetIndexId(), this-
>key_schema_, buffer_pool_manager));
    }
    else if(l_sum <= 8)
    {
      void *mem = heap_->Allocate(sizeof(BPlusTreeIndex<GenericKey<8>,
RowId, GenericComparator<8>>));
      return static_cast<Index*>(new(mem)BPlusTreeIndex<GenericKey<8>,
RowId, GenericComparator<8>>(this->meta_data_->GetIndexId(), this-
>key_schema_, buffer_pool_manager));
    }
    else if(l_sum <= 16)
    {
      void *mem = heap_->Allocate(sizeof(BPlusTreeIndex<GenericKey<16>,
RowId, GenericComparator<16>>));
      return static_cast<Index*>(new(mem)BPlusTreeIndex<GenericKey<16>,
RowId, GenericComparator<16>>(this->meta_data_->GetIndexId(), this-
>key_schema_, buffer_pool_manager));
    }
    else if(l_sum <= 32)
    {
      void *mem = heap_->Allocate(sizeof(BPlusTreeIndex<GenericKey<32>,
RowId, GenericComparator<32>>));
      return static_cast<Index*>(new(mem)BPlusTreeIndex<GenericKey<32>,
RowId, GenericComparator<32>>(this->meta_data_->GetIndexId(), this-
>key_schema_, buffer_pool_manager));
    }
    else if(l_sum <= 64)
    {
      void *mem = heap_->Allocate(sizeof(BPlusTreeIndex<GenericKey<64>,
RowId, GenericComparator<64>>));
      return static_cast<Index*>(new(mem)BPlusTreeIndex<GenericKey<64>,
RowId, GenericComparator<64>>(this->meta_data_->GetIndexId(), this-
>key_schema_, buffer_pool_manager));
    }
```

```
}
```

2.7 测试代码设计

测试场景	预期结果
创建新的CatalogManager(init = true)	重载后Index和Table数量均为0
重载已有的CatalogManager(init = false)	重载后Index和Table数量与退出前一致
Create Table（Table不存在）	Table数量+1，重载后对应的逻辑页码与初始化时一致
Create Table（Table已存在）	返回DB_TABLE_ALREADY_EXIST
Drop Table（Table存在）	Table数量-1，该表上的Index也一并删除
Drop Table（Table不存在）	返回DB_TABLE_NOT_EXIST
Get Table（Table存在）	TableInfo与创建时一致
Get Table（Table不存在）	返回DB_TABLE_NOT_EXIST
Get Tables	返回vector的size与实际表数量一致
Create Index（Table不存在）	返回DB_TABLE_NOT_EXIST
Create Index（Index已存在）	返回DB_INDEX_ALREADY_EXIST
Create Index（Index不存在）	Index数量+1，重载后对对应的逻辑页码与初始化时一致
Drop Index（Table不存在）	返回DB_TABLE_NOT_EXIST
Drop Index（Index不存在）	返回DB_INDEX_NOT_EXIST
Drop Index（Index存在）	Index数量-1
序列化反序列化	返回值一致， MAGIC_NUM与预期相等，信息与序列化前一致
Get Serialized Size	与序列化的返回值一致