模块 4 Catalog Manager

一. 概述

该模块负责管理、维护数据库的模式信息，主要包含两部分。

- 目录元信息
- 表和索引的管理

二. Catalog Manager 的实现

1. 元信息、序列化与反序列化

数据库中定义的表和索引在内存中分别包含其在定义时的元信息 table_meta_data 和 index_meta_data，为了将所有表和索引的定义信息持久化到数据库文件并在重启时从数据库文件中恢复，需要实现表和索引的元信息 TableMetadata 和 IndexMetadata 的序列化和反序列化操作。

在序列化时，需要为每一个表和索引都分配一个单独的数据页用于存储序列化数据，因此需要用于记录和管理这些表和索引的元信息被存储在哪个数据页中的数据对象 CatalogMeta，它的信息将会被序列化到数据库文件的第 CATALOG_META_PAGE_ID 号数据页中，默认值为 0。

```
class CatalogMeta {
 private:
  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_;
};
```

上图为 CatalogMeta 的数据成员，以 table_meta_pages 在序列化时，需要先写入其页数，然后用 MACH_WRITE_TO(Type, buf, DATA)进行序列化。

反序列化时，根据写入的页数用 MACH_READ_FROM(Type, buf)逐个取出。

```
class TableMetadata {
private:
  static constexpr uint32_t TABLE_METADATA_MAGIC_NUM = 344528;
  table_id_t table_id_;
  std::string table_name_;
  page_id_t root_page_id_;
  Schema *schema_;
};
```

上图为 TableMetadata 的数据成员，同样使用 MACH_WRITE_TO(Type, buf, DATA)函数进行序列化；string 类型的 table_name_需先获取其长度，再用 memcpy 将其长度与内容依次写入；schema_则调用 Schema 类里的 SerializeTo(char *buf)函数进行序列化。

反序列化时，需要定义一个临时字符数组 tmp 用于读取 table_name_，schema_需要调用 Schema 类里的 DeserializeFrom(char *buf, MemHeap *heap)函数进行反序列化。最后调用 TableMetaData 类中的 Create 函数分配空间并存储上述反序列化得出的信息。

```
class IndexMetadata {
private:
  static constexpr uint32_t INDEX_METADATA_MAGIC_NUM = 344528;
  index_id_t index_id_;
  std::string index_name_;
  table_id_t table_id_;
  std::vector<uint32_t> key_map_;  /** The mapping of index key to tuple key
*/
};
```

上图为 IndexMetadata 的数据成员，其序列化与反序列化操作与 TableMetadata 基本一致。序列化时，key_map_需要循环调用 MACH_WRITE_UINT32(buf, DATA)逐个写入；反序列化时，需要定义 key_map 向量，同样通过循环使用 vector 类的 push_back 函数读出并存放于其中。

2. 表和索引的管理

类 CatalogManager 在数据库实例（DBStorageEngine）初次创建时（init = true）初始化元数据；并在后续重新打开数据库实例时，从数据库文件中加载所有的表和索引信息。

```
class TableInfo {
 public:
  TableMetadata *table_meta_;
 private:
  TableHeap *table_heap_;
  MemHeap *heap_;
};

class IndexInfo {
private:
  IndexMetadata *meta_data_;
  Index *index_;
  TableInfo *table_info_;
  IndexSchema *key_schema_;
  MemHeap *heap_;
};
```

上图为 TableInfo 和 IndexInfo 类的数据成员，用于存放从数据库文件中加载出的表和索引信息，并且置于内存中。

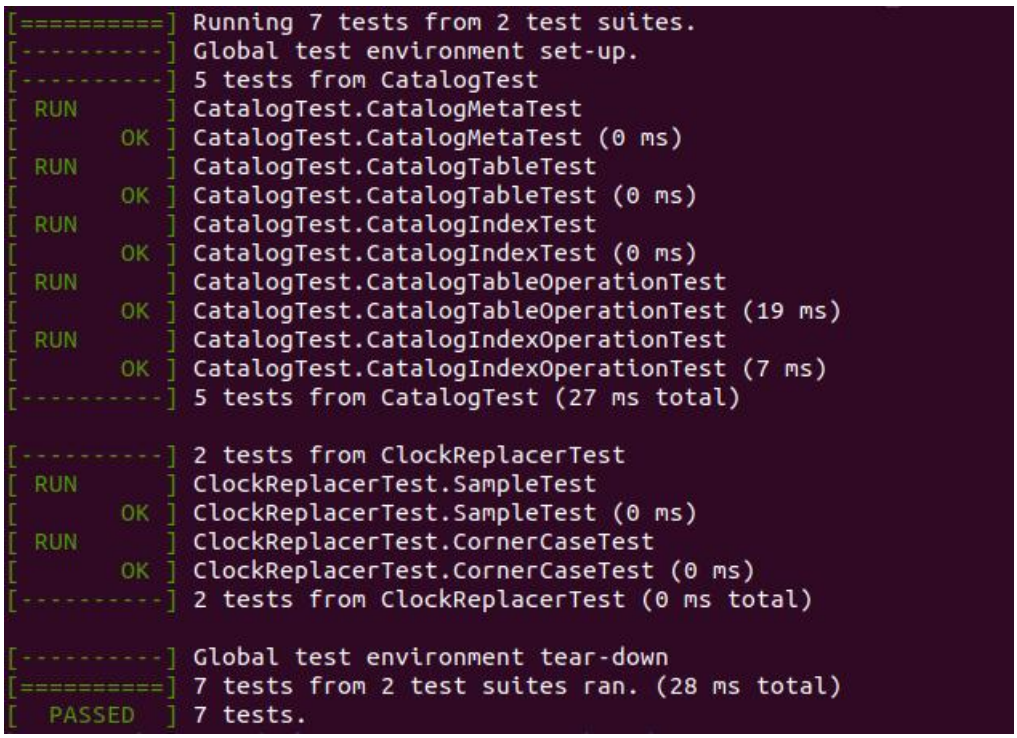
```
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema
    *schema, Transaction *txn, TableInfo *&table_info);

dberr_t CatalogManager::GetTable(const string &table_name,
    TableInfo *&table_info);

dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables);

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);

dberr_t CatalogManager::GetIndex(const std::string &table_name, const
    std::string &index_name, IndexInfo *&index_info);
```

```
dberr_t CatalogManager::GetTableIndexes(const std::string &table_name,
    std::vector<IndexInfo *> &indexes);

dberr_t CatalogManager::DropTable(const string &table_name);

dberr_t CatalogManager::DropIndex(const string &table_name,
    const string &index_name);

dberr_t CatalogManager::FlushCatalogMetaPage();

dberr_t CatalogManager::LoadTable(const table_id_t table_id,
    const page_id_t page_id);

dberr_t CatalogManager::LoadIndex(const index_id_t index_id,
    const page_id_t page_id);
```

以上函数承担操作数据库的功能，包括表和索引的创建、查找、加载、删除，将 CatalogMeta 强制写进磁盘的函数 FlushCatalogMetaPage()，在此不加赘述。

三. 测试结果

通过 test 文件夹下的 catalog_test.cpp 对 catalog 实现的正确性进行检验。



测试通过，运行无误。