**数据库个人报告**

3200102533 刘奕辰

**# 实现模块说明**

1. 共同完成了minisql结构的讨论和剖析，并共同参与完成了1.1，1.2,1.3模块的设计；
2. 负责完成1.5模块buffer pool manager的设计；
3. 负责整个模块4 catalog manager的设计；
4. 和组内另一位同学共同完成模块5 executor 的设计；

**# 模块功能详细介绍**

**#1 DISK MANAGER**

**#1.5 BUFFER POOL MANAGER**

Page \*BufferPoolManager::FetchPage(page\_id\_t page\_id)

该函数根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取。

该函数需要参数逻辑页码page\_id。

首先从内存中读取，若内存中已有该数据，直接返回数据页；若没有，访问磁盘。访问free\_list\_及replacer，若有空闲的位置，用其代替所求的数据页。其中，若该页上有内容，需要将内容写入磁盘防止数据丢失。从free\_list\_或replacer中删除该数据页并更新相应的元信息。

Page \*BufferPoolManager::NewPage(page\_id\_t &page\_id)

该函数分配一个新的数据页，并将逻辑页号于page\_id中返回。

该函数需要参数page\_id来储存分配的逻辑页号。

若free\_list或replacer中有空闲页，优先从free\_list中分配，更新相关页对应的元信息。若所有的页都已经被固定，直接返回空指针。

bool BufferPoolManager::DeletePage(page\_id\_t page\_id)

该函数将释放对应的数据页。

该函数需要参数page\_id表示将被释放的数据页。

查询page\_table\_，若不存在该数据页，直接返回true；若存在但已经被固定，返回false；若存在且未被固定，释放数据页，更新对应的元信息，将其插入free\_list\_。

bool BufferPoolManager::UnpinPage(page\_id\_t page\_id, bool is\_dirty)

该函数取消固定对应的数据页。

该函数需要参数page\_id表示将被取消固定的数据页，is\_dirty表示该数据页是否有内容。

若page\_table\_中不存在该数据页，返回false；若存在，将其pin\_count减1，调用replacer中的unpin( )函数进行取消固定，并根据is\_dirty设置pages\_对应的is\_dirty值。

bool BufferPoolManager::FlushPage(page\_id\_t page\_id)

该函数将数据页转存到磁盘中。

该函数需要参数page\_id表示需要被转存的数据页。

若page\_table\_中存在该数据页，调用磁盘中的WritePage( )函数写入磁盘。

**#4 CATALOG MANAGER**

**#4.1 模块概述**

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。

●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。

**#4.2 模块间调用关系**

调用了disk manager中的bufferpoolmanager,为每一个表和索引都分配一个单独的数据页；

调用了record manager中的column,row,schema的序列化和反序列化函数；

调用了index manager中的BplusTreeIndex,为表中的列创建索引；

被execute engine调用，执行上层模块中的内容；

**#4.3 模块函数实现**

**4.3.1 序列化相关操作**

**4.3.1.1 Catalogmeta**

**● CatalogMeta::SerializeTo(\*buf)**

该函数把记录索引对应的物理页和记录表对应的物理页两个map序列化储存到CATALOG\_META\_PAGE\_ID（第0页）中。序列化的时候首先分别储存两个页中元素的数量，接着再按顺序储存tableid/indexid和对应的pageid.

**● CatalogMeta::GetSerializedSize()**

该函数用于获取序列化的长度，根据两个map的大小计算，同时加上两个记录元素数量的整型大小即可。

**● CatalogMeta::DeserializeFrom(\*buf, \*heap)**

该函数将两个map从元信息页中反序列化出来。首先创建和元信息页相同类型的两个map，首先读取map中元素的数量，再一一读取元素加入这两个map中，接着利用这两个map和所提供的heap，构建新的catalogmeta。

**4.3.1.2 TablemetaData**

**● TableMetadata::SerializeTo(\*buf)**

该函数将tableinfo中的信息序列化到buf对应的空间中。Tableinfo中包含了tableid(表的序号），tablename(表的名字），rootpageid(表所储存的物理页id),和schema(表的信息）.序列化时要注意tablename为string类型，必须首先储存其长度，而schema序列化直接调用recordmanager中其序列化函数即可。

**● TableMetadata::GetSerializedSize()**

该函数获取对应的序列化长度，要加上储存tablename长度的整型长度。

**● TableMetadata::DeserializeFrom(\*buf, \*&table\_meta, \*heap)**

该函数调用了schema的反序列化函数，用从buf中读到的tableid,rootpageid,tablename和schema构建新的tablemetadata.其中读取tablename时不能用Mach\_read\_from而应该使用append函数。

**4.3.1.3 IndexmetaData**

**● IndexMetadata::SerializeTo(\*buf)**

该函数将indexmetadata中的内容序列到buf中。我们依次储存indexid,indexname,rootpageid和keymap.其中indexname和tablename一样需要额外一个整数来储存其长度。keymap是indexkey到tuplekey的映射，对应的整数是该列在表中的第几位,它同样需要一个整数来储存长度。同时indexname在储存时应当利用.c\_str()函数转化成char\*类型，否则在反序列化时会读取到乱码。

**● IndexMetadata::GetSerializedSize()**

该函数获取对应的序列化长度，要加上储存tablename和keymap的整型长度。

**● IndexMetadata::DeserializeFrom(\*buf, \*&index\_meta, \*heap)**

该函数是将信息从buf中读取出来并且生成新的indexmeta。我们依次读取indexid,indexname,rootpageid和keymap,接着利用读取到的这四个信息构建新的indexmeta.

**4.3.1.4 IndexInfo::Init(\*index\_meta\_data, \*table\_info, \*buffer\_pool\_manager)**

该函数将反序列化得到的metadata和tableinfo生成Indexinfo.其中，key\_schema的生成需要使用到schema中的shallowcopyschema函数。该函数中三个参数分别是tableinfo中的schema，keymap是用indexmeta中的keymap生成的，利用getkeymapping函数获得。index需要调用Bufferpoolmanager和bplustreeindex生成索引。其中key的长度需要在schema.h中加入函数统计各列中key的长度总和，选择对应长度的key.

加入的函数getmaxlen定义如下：

inline uint32\_t GetMaxLen() const {

uint32\_t res=0;

for(auto column:columns\_)

res+=column->GetLength();

return res;

}

**4.3.2 数据库相关操作**

**4.3.2.1 变量说明**

[[maybe\_unused]] std::atomic<table\_id\_t> next\_table\_id\_;

用来储存下一张表的id，这样分配tableid时可直接使用

[[maybe\_unused]] std::atomic<index\_id\_t> next\_index\_id\_;

用来储存下一张表的id，这样分配indexid时可直接使用

std::unordered\_map<std::string, table\_id\_t> table\_names\_;

用来储存表的名字及其对应的id

std::unordered\_map<table\_id\_t, TableInfo \*> tables\_;

用来储存表的信息和对应的id

std::unordered\_map<std::string,std::unordered\_map<std::string,index\_id\_t>> index\_names\_;

用来储存表的信息和表上面对应的id名字及其对应id编号

[[maybe\_unused]] std::unordered\_map<index\_id\_t, IndexInfo \*> indexes\_;

用来储存索引的名字及其对应的id

**4.3.2.2 函数介绍**

CatalogManager::CatalogManager(BufferPoolManager\*buffer\_pool\_manager, LockManager \*lock\_manager,LogManager \*log\_manager, bool init)

该函数分为两种情况，一种是（init=true)首次创建数据库，另一种是(init=false)重新加载数据库。

首次调用数据库非常简单，只要调用heap初始化catalogmanager即可；

重新加载数据库时，首先要将catalogmeta进行反序列化得到元信息，这是生成所有其他信息的必备条件；接着我们遍历catalogmeta中的两个map，从pageid中分别反序列化tablemetadata和indexmetadata，构建对应的tableinfo和indexinfo,再加入到catalogmanager的几个map中，从而还原出完整的catalogmanager信息。

dberr\_t CreateTable(const std::string &table\_name, TableSchema \*schema, Transaction \*txn, TableInfo \*&table\_info)

该函数的作用是利用信息创建一张表，生成对应的tableinfo.tableinfo由tablemetadata和tableheap两部分构成，这两部分都需要利用所给信息，调用对应的构造函数。其中，

tablemetadata的rootpageid就是我们为该表所分配的数据页。我们需要调用bufferpoolmanager的newpage函数获取一个对应的空闲页。同时，我们将nexttableid作为新表的tableid，同时将tableid加一为下一张表做准备。调用tableinfo的Init函数生成对应信息。此外，我们要获取空闲页的数据内容buf作为序列化的空间。调用tablemetadata的序列化函数。此外，由于catalogmeta的table\_meta\_pages也发生了改变，我们调用fetchpage，清空元信息页并重新序列化，以保证反序列化时读取的是正确信息。

dberr\_t GetTable(const std::string &table\_name, TableInfo \*&table\_info)

该函数根据tablename得到对应的tableinfo。我们只要在table\_names里面找到对应的tableid，然后根据tableid在tables\_中获取对应的tableinfo即可。

dberr\_t GetTables(std::vector<TableInfo \*> &tables) const

该函数相当于将所有表中的信息储存在一个vector里面。只要遍历vector生成即可。

dberr\_t CreateIndex(const std::string &table\_name, const std::string &index\_name,

const std::vector<std::string> &index\_keys, Transaction \*txn,

IndexInfo \*&index\_info);

该函数是利用信息在表上生成索引。和createtable一样，首先都要调用bufferpoolmanager来生成对应的空闲页。需要注意的是，所给信息中index\_keys是字符串类型的vector,而我们需要的是int类型的keymap.因此我们需要调用getcolumnindex函数将Index\_keys转换为keymap，再构建indexmetadata.此外，要注意一个table可以对应多个index，因此index\_names插入时最好采用数组元素赋值的方式。最后同样需要清空元信息页中的内容并重新序列化。

dberr\_t GetIndex(const std::string &table\_name, const std::string &index\_name, IndexInfo \*&index\_info) const

该函数通过表和索引的名字获取对应的索引信息。只要在index\_names\_中寻找对应的indexid,接着在Indexes\_表中寻找即可。

dberr\_t GetTableIndexes(const std::string &table\_name, std::vector<IndexInfo \*> &indexes) const

该函数获取一张表上的所有index信息。寻找方法和gettables相同。

dberr\_t DropTable(const std::string &table\_name)

该函数根据表的名字删除对应的表。要到catalogmanager所有储存该表信息的map中进行删除，同时也要在catalogmeta中进行删除。由于catalogmeta发生改变，我们必须再次清空元信息页重新序列化。此外储存该表信息的页面应当调用deletepage函数进行回收，以免造成页面的浪费。

dberr\_t DropIndex(const std::string &table\_name, const std::string &index\_name)

该函数根据对应的表名字和索引名字删除索引。同样需要改变catalogmeta;同样也需要调用deletepage回收页面。

dberr\_t FlushCatalogMetaPage() const

该函数没有在我们的catalog manager中被使用。我认为其作用是将catalogmeta中所有调用的页面转存到磁盘当中。

**#5 EXECUTOR\_ENGINE**

ExecuteEngine::selectrow(pSyntaxNodeast, std::vector<Row\*>& r, TableInfo\* t, CatalogManager\* c)\*\*

根据传入的数据，判断指令中的逻辑“and”，“or”等，然后将符合条件的table中的rows保存在容器中。

然后根据传入的指令的要求，如“=”， “>”, "<>"等逻辑，再将上一步中满足的rows进一步筛选，保存满足条件的rows。在“=”类型时，应该首先判断对应的列有无创建索引，如果有应该通过索引的方式来寻找。

值得注意的是，要根据char, int, float 类型，分别进行上述的操作。

ExecuteEngine::ExecuteDelete(pSyntaxNode ast, ExecuteContext \*context)\*\*

该函数主要是从语义树中第一个结点中给出的数据库的名字，然后继续根据语义树，读取要创建的table的名字，和table中的values的相关数据。

然后调用第四章中实现的模块的函数，将这行row从table中删除。

同时更新现有的index索引，将这条记录的所有indexes删除。

ExecuteEngine::ExecuteUpdate(pSyntaxNode ast, ExecuteContext \*context)\*\*

该函数主要是从语义树中第一个结点中给出的数据库的名字，然后继续根据语义树，读取要创建的table的名字，和table中的values的相关数据。

然后调用第四章中实现的模块的函数，将table中的这条row进行更新。

同时更新现有的index索引，将这条记录的所有indexe更新。

**# 收获与建议**

本次数据库设计实验花费了我们大量时间，占用了整个夏学期接近一半的时间来完成任务。通过小组成员齐心协力，我们最终顺利完成了Minisql实验并在验收中通过了所有测试点。虽然辛苦但是我受益匪浅。

我认为这个minisql的框架设计的非常精巧，让我们在编程同时，加深了对数据库架构的理解，同时也通过揣摩每个函数的作用懂得了不同模块之间的相互调用关系。此外，对map,vector的频繁使用，也让我们c++编程能力大幅提升。

个人的一点建议，一方面是由于数据库实验工程量较大，从春学期末开始布置较为合适；此外，对于架构的设计，我认为2，4模块有相似之处，因而更适合将3，4调换，将更难的索引设计b+树放到后面；此外，对于char类型的字符，在储存过程中末尾容易出现乱码，这可能是需要改进的地方。

总而言之，我在数据库设计的实验中提高了团队协作意识和动手能力，为以后从事相关工作打下了坚实基础。