2021 春《数据库系统》实验报告

实验 3：查询执行器

姓名： 卢兑玩　　学号：L170300901　　班级：1803501

1. 实验目的

在实验 2 实现的 BadgerDB 缓冲池管理器(buffer pool manager)的基础上，本次实验继续实现 BadgerDB 的存储管理器(storage manager)和查询执行器(query executor)，具体完成以下内容：

使用堆文件(heap file)存储关系，实现向关系中插入(insert)元组和删除(delete)元组的功能。

实现自然连接(natural join)操作算法，对两个关系进行自然连接，具体实现一趟连接(One-Pass Join)、基于块的嵌套循环连接(Block-based Nested Loop Join)、Grace 哈希连接(Grace Hash Join) 等算法。Grace 哈希连接算法的实现为选作。

2. 实验准备

见实验指导书。根据实验指导书来完成。

3. 实验内容

在本次实验中，你需要实现 TableSchema, HeapFileManager, OnePassJoinExecutor, NestedLoopJoinOperator 类中的若干方法，完成 BadgerDB 的存储管理和查询执行功能。具体需要实现的方法如下：

```
// schema.cpp

TableSchema* TableSchema::fromSQLStatement(const string& sql);

void TableSchema::print() const;

// storage.cpp

RecordId HeapFileManager::insertTuple(const string& tuple, File& file, BufMgr*
bufMgr);

void HeapFileManager::deleteTuple(const RecordId& rid, File& file, BufMgr*
bugMgr);

string HeapFileManager::createTupleFromSQLStatement(const string& sql, const
Catalog* catalog);

// executor.cpp

void TableScanner::print() const;
```

```
TableSchema JoinOperator::createResultTableSchema(
        const TableSchema& leftTableSchema,
        const TableSchema& rightTableSchema);

bool OnePassJoinOperator::execute(int numAvailableBufPages, File& resultFile);

bool NestedLoopJoinOperator::execute(int numAvailableBufPages, File&
resultFile);
```

schema .cpp

（1）

```
vector <string> split(const string& cutting, string sign)
```

首先在 schema.h 中添加一个 split 函数的声明，用来按照指定字符切割字符串：返回 vector 类型变量：

```
vector <string> split(const string& cutting, string sign);
```

然后再在 schema.cpp 中实现 split，输入要切割的字符串 cutting 和切割字符 sign，遍历 cutting，如果碰到 sign，则在 vector 遍历中加入到此符号之前一位的子串，最后得到分割结果。

(2)
```
TableSchema TableSchema::fromSQLStatement(const string& sql)
```

首先将输入的 SQL 语句按照空格“ ”分割，则 table 的名字即为分割后得到的 vector 变量的第三个值

```
vector<string> aName = split(sql, " ");
tableName = aName[2];
```

，然后如果名字中包含#着##，则为临时表

```
if(aName[2].find("#",0)!=aName[2].npos || aName[2].find("##",0)!=aName[2].
  isTemp = true;
```

。之后找到前后小括号的位置，得到属性部分：

```
unsigned int froS = sql.find_first_of("(");
unsigned int backS = sql.find_last_of(")");
const string attrsString = sql.substr(froS + 1, backS - (froS + 1));
```

将属性按照“,”分割，即可得到各个属性值的定义，遍历属性 vector 对所有属性值进行如下操作：按照空格分割，vector 中第一个即为属性名字。再使用 find 找到属性的类型，并分类进行最大值的获取

```
if(tmp.find("INT") != tmp.npos){
    attrType = INT;
    maxSize = 4;
}
```

```
if(tmp.find("CHAR") != tmp.npos){
    attrType = INT;
    int t = attrName[1].length();
    string S = attrName[1].substr(5, t-6);
    maxSize = atoi(S.data());
}
 if(tmp.find("VARCHAR") != tmp.npos){
     attrType = INT;
     int t = attrName[1].length();
     string S = attrName[1].substr(8, t-9);
     maxSize = atoi(S.data());
 }
```

然后再用 find 看属性是否是 NULL、UNIQUE 的即可。

(3)
```
void TableSchema::print() const
```

输出表名、是否为临时表、属性总数量、属性名字、类型、最大值、是否为 NULL，是否 UNIQUE。

storage.cpp

(1)
```
RecordId HeapFileManager::insertTuple(const string& tuple,
                                      File& file,
                                      BufMgr* bufMgr) {
```

要插入元组，就要知道可用的 recordid。

首先，运用文件的迭代器遍历文件的页

```
for(FileIterator i = file.begin(); i != file.end(); ++i){
```

在遍历中，首先根据迭代器得到遍历到的页的 id，然后运用缓冲池的 readpage 方法得到页对象。如果空间充足，则运用 insertRecord 方法得到插入的 recordid，将缓冲池的此页设置为 dirty。

```
recordId = P->insertRecord(tuple);
bufMgr->unPinPage(&file, pageId, true);
```

一旦发现页号为无效值或者无充足空间，则跳出遍历从缓冲池申请页来插入元组

```
bufMgr->allocPage(&file, pageId, P);
```

，并根据页的 insertRecord 方法得到 recordid，再将缓冲池的此页设置为 dirty 即可。

(2)

```
void HeapFileManager::deleteTuple(const RecordId& rid,
                                  File& file,
                                  BufMgr* bugMgr) {
```

删除元组，已知 recordid，则找到所在页即可，且 recordid 包含 page_number，所以直接运用缓冲池的 readpage 方法得到页对象，运用页的 deleteRecord 方法即可删除元组

```
P->deleteRecord(rid);
```

(3)

```
string BitsSequence(char* attInfo, int len)
```

由于实验规定元组传递时传递的不是字符串，而是字节序列，所以首先设定一个将元组字符串转换成字节序列的方法。arrInfo 参数表示 insert 的属性值，len 表示属性值的 maxsize。

声明一个字符串流类型的参数 ans，然后循环遍历长度 len（i）

```
for (int i = 0; i < len; i++)
```

，内套循环遍历长度 char 类型的长度*8（j）

```
for (int j = 0; j < sizeof(char)*8; j++)
```

，表示 bit 数如果 arrInfo 的第 i 位不是‘\0’，那么则将它与 0x80 作位与操作，0x80 为 10000000，每内循环一次 0x80 则往右移 j 位，目的在于逐位提取 attInfo 的 bit 值

```
if(*(attInfo + i) & (0X80 >> j))
```

如果到了尽头，则后面加 X 即可。最终返回 ans 的 string 值。

(4)

```
string HeapFileManager::createTupleFromSQLStatement(const string& sql,
                                                     const Catalog* catalog)
```

首先声明一个变量 bitSequence，用来存储元组的字节序列。跟分析 create 语句一样，首先根据“ ”分离字符串，然后得到的 vector 变量的第三个值即为要插入元组的表的名字，然后根据名字得到 tableid

```
TableId tId = catalog->getTableId(tableName);
```

，之后将 tableid 写入 bitSequence。然后用跟之前同样的方法获得属性值，之后将属性值根据“,”分割后得到不同属性的值。遍历不同属性的值，根据值从表模式中得到属性的 maxsize

```
int MaxSize = tSchema.getAttrMaxSize(index);
```

，再将 maxsize 值加到

bSeqSize 变量中，然后此时用之前的 BitsSequence 方法获取 bits 序列 `string valueTmp = BitsSequence((char*)tmp.data(), MaxSize+1);`，再加入到变量中 attrsBitsSeq 中，为后来加入 bitSequence 变量做准备。

遍历完成后，用之前的 bSeqSize 变量再加上 64，代表元组头（id 和 size）的长度，将其加入到 bitSequence 变量中，最后再将 attrsBitsSeq 加入 bitSequence 变量即可得到最终答案。

executor.cpp

首先设置几个全局变量：`static string JoinAttr`表示进行连接的属性，即两个表中的共有属性；`:multimap<string,string> JoinAttrs;`存储共有属性值--含有该值的元组的键值对。

(1) `string translateBitsSeq(const string& bitsSeq)`

进行元组的 bitSequence 的翻译。首先遍历得到的 bitSequence，将空位补充的 X 都给去掉。然后将剩下的序列遍历，遍历长度为序列长度/8：

```
for (int i = 0; i < attrV.length()/8; i++){
for (int j = 7; j >= 0; j--){
if(attrV[j + i*8] == '1'){
    tmp += pow(2.0, index);
}
char Attrvalue = char(tmp);//Decimal to character
```

，内部嵌套一个循环，目的是从最低位开始算二进制数的值。最后再将这个值转换为字符即可。

(2) `bool compareLandR(File L,File R)`

比较两个文件的页数多少。运用迭代器对L和R文件进行迭代，并设置参数累加计算页数，最终，如果L文件页数少于等于R，则返回true，否则返回false

(3) `void TableScanner::print() const`

打印数据库。首先获取tableFile和缓冲池，然后刷一遍文件 `buf->flushFile(&tableFile);`，再利用file的迭代器进行迭代完成数据库表的读取。获取并打印表模式的操作与schema.cpp中的print函数相同。之后根据迭代器获得pageid `PageId pageBeginId = (*iterFile).page_number();` 进行while循环，pageid等于0时跳出。根据缓冲池的readpage方法得到page的对象，获取pageid和page中的slotid `PageId pageid = page->page_number();` `SlotId slotid = page->begin().getNextUsedSlot(0);` 再进行一个 while 循环，slotid 等于 0 时跳出。根据 pageid 和 slotid 构建 recordid `RecordId recordid = {pageid, slotid};` 之后通过 recordid 获得元组字节序列 `string bitsSeq = page->getRecord(recordid);`，经过翻译后遍历输出即可。

(4)

```
JoinOperator::JoinOperator(const File& leftTableFile,
                           const File& rightTableFile,
                           const TableSchema& leftTableSchema,
                           const TableSchema& rightTableSchema,
                           Catalog* catalog,
                           BufMgr* bufMgr)
```

这个方法为构建连接后表模式的操作，调用createResultTableSchema即可，当leftfile更小时createResultTableSchema参数为（left，right），否则反过来即可。

(5)

```
TableSchema JoinOperator::createResultTableSchema(
    const TableSchema& leftTableSchema,
    const TableSchema& rightTableSchema) {
```

这个方法为构建连接后表模式的操作。首先根据自然连接的属性可知要先获取左右表的属性都有哪些，再从中找到共有属性，再进行连接即可。思路为先将其中一个表的全部属性放入

```
for (int i = 0; i < numL; i++){
  string name = leftTableSchema.getAttrName(i);
  DataType attrType = leftTableSchema.getAttrType(i);
  int maxSize = leftTableSchema.getAttrMaxSize(i);
  bool isNotNull = leftTableSchema.isAttrNotNull(i);
  bool isUnique = leftTableSchema.isAttrUnique(i);
  Attribute tmp1((string &)name, (DataType &)attrType, maxSize, isNotNull,
  attrs.push_back(tmp1);
}
```

，再遍历另一个表的属性，如果与某个已经加入的属性相等

```
if((string &)t.attrName == (string &)rightTableSchema.getAttrName(i)){
```

，则这个属性不再加入，且这个属性即为共有属性，把这个属性的名字付给一开始声明的全局变量。这样即可完成这个方法。最终

```
return TableSchema("JoinResultTABLE", attrs, true);
```

(6)

```
void writeToBuf(Page* page, BufMgr* buf, const TableSchema& tableSche)
```

该方法将表模式写入缓冲池中并记录共有属性值--含有该值的元组的键值对。首先根据page对象获得pageid和起始slotid

```
PageId pageid = page->page_number();
SlotId slotid = page->begin().getNextUsedSlot(0);
```

，然后获取属性总数并进行遍历，找到两个表共有属性的下标

```
if(tableSche.getAttrName(i) == JoinAttr){
    JoinAttrNum = i;
}
```

，再进行 while 循环，当 slotid 等于 0 时跳出。循环中首先构建 recordid，然后以此获得元组的字节序列

```
string bitsSeq = page->getRecord(recordid);
```

，翻译成字符串之后进行分割操作，遍历将属性值依次写入字符串流变量中

```
string A = *i;
attrStr[index - 2] = A;
attrV<<A<<",";
```

，最后将共有属性值--连接元组的键值对写入全局变量 JoinAttrs 中即可

```
JoinAttrs.insert(make_pair(attrStr[JoinAttrNum], attrV.str()));
```

(7) `int join(File file,Page* page,const TableSchema& resultable,const TableSchema& tableSche, Catalog *catalog,BufMgr* bufMgr)`

目的在于将输入的page与缓冲池中的数据进行连接，并返回连接表中元组的个数。已知page对象，可以根据page对象获得起始slotid和pageid，然后获取属性总数并进行遍历，

```
if(tableSche.getAttrName(i) == JoinAttr){
    JoinAttrNum = i;
}
```

找到两个表共有属性的下标 再进行 while 循环，当 slotid 等于 0 时跳出。循环中首先构建 recordid，然后以此获得元组的字节序列 `string bitsSeq = page->getRecord(recordid);`，翻译成字符串之后进行分割操作，遍历将属性值依次写入字符串流变量中

```
string A = *i;
attrStr[index - 2] = A;
attrV<<A<<",";
```

。先用 count 方法看此共有属性值是否在 JoinAttrs 中 `if(JoinAttrs.count(tmp) != 0){`，若在的话就继续如下操作：迭代遍历所有 key 值为此共有属性值的键值对，将它的值写入一 stringstream 变量，然后再把之前得到的页中的元组值写入

```
string s = tupleAvail->second;
stringstream atr_value_result;
atr_value_result<<s<<attrV.str();
```

，然后这里便得到了 insert 语句的属性值部分，再构建 insert 语句 `result<<"INSERT INTO " <<resultable.getTableName()<< " VALUES (" << atr_val`，运用之前实现的 HeapFileManager::createTupleFromSQLStatementinsertTuple 和方法即可把该元组插入新的连接表中

```
string tuple = HeapFileManager::createTupleFromSQLStatement(result.str(), ca
HeapFileManager::insertTuple(tuple, file, bufMgr);
```

，再把连接表中元组的个数加一即可。

最后，再将新数据从缓冲池刷到文件中即可 `bufMgr->flushFile(&file);//Write result file`

(8) `bool OnePassJoinOperator::execute(int numAvailableBufPages, File& resultFile)`

首先将连接结果表模式和表名加入目录中 `catalog->addTableSchema(resultTableSchema,resultTableSchema.getTableName());`，然后声明变量分别为左表文件、右表文件、左表模式和右表模式。然后对比大小，以进行下面的操作。为了不影响之后的操作，首先清空全局变量 JoinAttrs。然后用迭代器迭代较小的表文件，遍历将该表文件所有的页都读入到缓冲池中 `buf->readPage(&fileR, startIdR, p);`，再将读写数加 1： `numIOs++;`，然后把其写入缓冲池中并记录共有属性值--含有该值的元组的键值对。将缓冲池已使用页数加1： `numUsedBufPages++;`。之后再用迭代器遍历较大的表文件，将其中的页依次读入缓冲池，并将此页与较小表文件的每一页进行连接操作 `join(resultFile, p, resultTableSche, tableScheL, catalog, buf);`，这样即可完成一次连接操作。

(9) `ol NestedLoopJoinOperator::execute(int numAvailableBufPages, File& resultFile)`

首先将连接结果表模式和表名加入目录中

```
catalog->addTableSchema(resultTableSchema,resultTableSchema.getTableName());
```

，然后声明变量分别为左表文件、右表文件、左表模式和右表模式。然后对比大小，以进行下面的操作。根据基于块的嵌套循环连接的原理可知，需要将较小的文件的 M-1 块写入缓冲池，M 即为缓冲池中可用页数 `int numUsedR = numAvailableBufPages - 1;` 。为了不影响之后的操作，首先清空全局变量 JoinAttrs。然后循环遍历较小的表文件，将其中的 M-1 个页写入缓冲池 `for (int i = 0; i < numUsedR`

```
writeToBuf(p,bufMgr,tableScheR);
```

，再在这个循环内部用迭代器遍历较大的表文件

```
FileIterator iterL = fileL.begin();
PageId startid = (*iterL).page_number();
while(startid != 0){
```

，将其中的页依次读入缓冲池，并将此页与较小表文件的这一轮循环所写入的页进行连接操作

```
join(resultFile, p, resultTableSche, tableScheL, catalog, buf);
```

，这样即可完成基于块的嵌套循环连接操作。

4. 实验结果

打印数据库模式：

```
TableName is : r
Is it TempTable : FALSE
The number of attribute is : 2
```

打印数据库：

```
The number of attribute is : 2
#--------------#--------------#
        a              b
#--------------#--------------#
        r0             0
        r1             1
        r2             2
        r3             3
        r4             4
        r5             5
        r6             6
        r7             7
        r8             8
        r9             9
        r10            10
        r11            11
        r12            12
        r13            13
        r14            14
        r15            15
        r16            16
        r17            17
        r18            18
```

一次连接结果：

```
Test One-Pass Join ...
# Result Tuples: 500
# Used Buffer Pages: 4
# I/Os: 16
TableName is : JoinResultTABLE
Is it TempTable : TRUE
The number of attribute is : 3
#--------------#--------------#--------------#
        b              c              a
#--------------#--------------#--------------#
        0              s0             r00
        1              s1             r11
        2              s2             r22
        3              s3             r33
        4              s4             r44
        5              s5             r55
        6              s6             r66
        7              s7             r77
        8              s8             r88
        9              s9             r99
        10             s10            r1010
        11             s11            r1111
        12             s12            r1212
```

嵌套连接结果：

```
Test Nested-Loop Join ...
# Result Tuples: 500
# Used Buffer Pages: 4
# I/Os: 16
TableName is : JoinResultTABLE
Is it TempTable : TRUE
The number of attribute is : 3
#--------------#--------------#--------------#
        b              c              a
#--------------#--------------#--------------#
        0              s0             r00
        1              s1             r11
        2              s2             r22
        3              s3             r33
        4              s4             r44
        5              s5             r55
        6              s6             r66
        7              s7             r77
        8              s8             r88
        9              s9             r99
        10             s10            r1010
        11             s11            r1111
```

测试结果：

```
#--------------#--------------#--------------#
Test Completed
```

5. 总结与体会

通过这次实验，我对于磁盘上数据库的操作细节有了更加清晰的认识，有助于我更深入地了解具体步骤。同时我还对几种连接方式进行了具体实现，对上课内容的理解有了更深的认识。这次实验设计的非常好。如果能再对所有需要用到的已实现方法进行介绍就更好了。