实验题目	查询执行器实现			实验日期	2022.05.01
班级	1903103	学号	1190200523	姓名	石翔宇

CS33503 数据库系统实验

实验检查记录

实验结果的正确性(60%)		表达能力(10%)	
实验过程的规范性(10%)		实验报告(20%)	
加分(5%)		总成绩(100%)	

实验报告

一、实验目的

1. 掌握各种关系代数操作的实现算法，特别是连接操作的实现算法。
2. 在实验 2 完成的缓冲区管理器的基础上，使用 C++面向对象程序设计方法实现查询执行器。

二、实验环境

硬件设备：Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
软件系统：Windows 11 22H2、Ubuntu 20.04.4 LTS
开发工具：Visual Studio Code 1.65.2

三、实验过程

实现方法：

我们实现了基于块的自然连接执行器，下面我们将按照代码执行步骤来依次介绍实现细节：

1.
```
badgerdb::TableId leftTableId = catalog->getTableId("r");
badgerdb::TableId rightTableId = catalog->getTableId("s");
badgerdb::File left = File::open(catalog->getTableFilename(leftTableId));
badgerdb::File right = File::open(catalog->getTableFilename(rightTableId));
```

这部分代码从传入的 catalog 提取待连接的两个表，以及两个表所存在的两个文件的对象，即 left 和 right。

2.
```
int leftForeignKeyId, rightForeignKeyId;
vector<int> rightOnlyAttrIds;
for (int j = 0; j < rightTableSchema.getAttrCount(); j++) {
    bool rightOnly = true;
    for (int i = 0; i < leftTableSchema.getAttrCount(); i++) {
        if ((leftTableSchema.getAttrName(i) == rightTableSchema.getAttrName(j)) && (leftTableSchema.getAttrType(i) == rightTableSchema.getAttrType(j))) {
            leftForeignKeyId = i;
            rightForeignKeyId = j;
            rightOnly = false;
        }
    }
    if (rightOnly)
        rightOnlyAttrIds.push_back(j);
}
```

leftForeignKeyId 和 rightForeignKeyId 分别表示左右两个表相同属性的位置，我们用下面的两个 for 循环代码来获取他们，用(leftTableSchema.getAttrName(i) == rightTableSchema.getAttrName(j)) && (leftTableSchema.getAttrType(i) == rightTableSchema.getAttrType(j))表示

两个属性完全相同。

除了上面的两个变量，我们还维护了一个可变长数组 rightOnlyAttrIds，表示只在右边表中出现的属性的位置，以方便下面代码的执行。

```
vector<badgerdb::Page*> bufferedLeftPages;
badgerdb::FileIterator leftPage = left.begin();

while (leftPage != left.end()) {
    while (leftPage != left.end() && bufferedLeftPages.size() < numAvailableBufPages - 1) {
        badgerdb::Page* bufferedLeftPage;
        bufMgr->readPage(&left, (*leftPage).page_number(), bufferedLeftPage);
        numIOs += 1;
        bufferedLeftPages.push_back(bufferedLeftPage);
        leftPage++;
    }
```

3.

由于基于块的嵌套连接算法首先要读入 N-1 个页，我们定义了变长数组 bufferedLeftPages，我们用第二行这个 while 来读入这些页。

```
    for (int index = 0; index < bufferedLeftPages.size(); index++) {
        badgerdb::Page* bufferedLeftPage = bufferedLeftPages[index];

        for (badgerdb::FileIterator rightPage = right.begin(); rightPage != right.end(); rightPage++) {
            badgerdb::Page* bufferedRightPage;
            bufMgr->readPage(&right, (*rightPage).page_number(), bufferedRightPage);
            numIOs += 1;

            for (badgerdb::PageIterator leftRecord = bufferedLeftPage->begin(); leftRecord != bufferedLeftPage->end(); leftRecord++) {
                vector<string> leftInfo = createDataFromTuple(*leftRecord, leftTableSchema);
                numUsedBufPages += 1;

                for (badgerdb::PageIterator rightRecord = bufferedRightPage->begin(); rightRecord != bufferedRightPage->end(); rightRecord++) {
                    vector<string> rightInfo = createDataFromTuple(*rightRecord, rightTableSchema);
                    numUsedBufPages += 1;

                    if (leftInfo[leftForeignKeyId] == rightInfo[rightForeignKeyId]) {
                        string sql = "INSERT INTO TEMP_TABLE VALUES (" + leftInfo[0];
                        for (int i = 1; i < leftTableSchema.getAttrCount(); i++) {
                            sql += ", " + leftInfo[i];
                        }
                        int rightOnlyAttrIdsSize = rightOnlyAttrIds.size();
                        for (int i = 0; i < rightOnlyAttrIdsSize; i++) {
                            sql += ", " + rightInfo[rightOnlyAttrIds[i]];
                        }
                        sql += ");";

                        string tuple = HeapFileManager::createTupleFromSQLStatement(sql, catalog);
                        HeapFileManager::insertTuple(tuple, resultFile, bufMgr);
                        numResultTuples += 1;
                    }
                }
            }
            bufMgr->unPinPage(&right, (*rightPage).page_number(), false);
        }
        bufMgr->unPinPage(&left, bufferedLeftPage->page_number(), false);
    }
    bufferedLeftPages.clear();
```

4.

在读入 N-1 个页后，用 for 语句遍历左右边的表，再遍历表中的每一行元组。如果当前元组的相同属性的值相同，则连接这两个元组。我们书写 SQL 语句，首先将左边表的所有属性写入，再按照之前得到的 rightOnlyAttrIds 来将只在右边表出现的属性写入。

实验题目	查询执行器实现			实验日期	2022.05.01
班级	1903103	学号	1190200523	姓名	石翔宇

```
vector<string> createDataFromTuple(string record, badgerdb::TableSchema schema) {
    vector<string> ret;
    int prev = 0;
    for (int i = 0; i < schema.getAttrCount(); i++) {
        DataType dataType = schema.getAttrType(i);

        switch (dataType) {
        case INT: {
            int value = 0;
            for (int j = 0; j < 4; j++)
                value = (value << 8) + record[prev + j];
            ret.push_back(to_string(value));
            prev += 4;
            break;
        }
        case CHAR: {
            int length = schema.getAttrMaxSize(i);
            ret.push_back(record.substr(prev, length));
            prev += length + (4 - (length % 4)) % 4;
            break;
        }
        case VARCHAR: {
            int length = record[prev];
            prev += 1;
            ret.push_back(record.substr(prev, length));
            prev += length + (4 - ((length + 1) % 4)) % 4;
            break;
        }
        }
    }

    return ret;
}
```

5. 上面的代码还调用了我们新添加的函数 createDataFromTuple。这个函数旨在将内部存储元组的字节序列转换为各个属性的值。

实验结果：

测试中的左右边表的元组结构分别如下所示：

(r2000000, 0)　　(0, s0)

程序输出形式为：

(r0000000, 0, s0)

```
Test Nested-Loop Join ...
# Result Tuples: 500
# Used Buffer Pages: 50500
# I/Os: 4
(r0000000,0,s0)
(r1000000,1,s1)
(r2000000,2,s2)
(r3000000,3,s3)
(r4000000,4,s4)
(r5000000,5,s5)
(r6000000,6,s6)
(r7000000,7,s7)
(r8000000,8,s8)
(r9000000,9,s9)
(r1000000,10,s10)
(r1100000,11,s11)
(r1200000,12,s12)
(r1300000,13,s13)
(r1400000,14,s14)
(r1500000,15,s15)
(r1600000,16,s16)
(r1700000,17,s17)
```

如图所示，实现的实验代码正确完成了自然连接的任务，完成了实验。

四、实验结论

实验题目	查询执行器实现			实验日期	2022.05.01
班级	1903103	学号	1190200523	姓名	石翔宇

通过本次对查询执行器的实现，我更深刻地了解了数据库管理系统的查询执行器的工作原理，体会到了数据库管理系统中独特的魅力。也通过使用C++面向对象程序设计方法实现了查询执行器，对面向对象程序设计方法有了更深的理解。