实验4: 查询执行器实现(2022春)

主讲教师: 邹兆年(znzou@hit.edu.cn)

1 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3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。

2 实验准备

在实验3完成的BadgerDB缓冲区管理器的基础上，本实验继续实现BadgerDB的查询执行器。本实验提供了关系模式定义(relation schema definition)、系统目录(system catalog)、存储管理器(storage manager)、查询执行器(query executor)等DBMS组件的C++声明及大部分实现。请在进行本次实验编程前，了解这些组件的声明及定义。

2.1 关系模式定义(Relation Schema Definition)

为了记录关系模式，我们在schema.h和schema.cpp中声明和定义了相关的数据类型和类。

2.1.1 DataType数据类型

BadgerDB支持三种SQL标准数据类型：

- INT：整型
- CHAR(n)：定长字符串型
- VARCHAR(n)：变长字符串型

这三种数据类型在BadgerDB内部的定义如下：

```
/**
 * Data type definitions: INT, CHAR(n), VARCHAR(n)
 */
enum DataType { INT, CHAR, VARCHAR };
```

2.1.2 Atrribute类

本实验定义了Attribute类，用Attribute类的对象记录关系属性(attribute)的定义。Attribute类的声明如下：

```
/**
 * Attribute definition
 */
class Attribute {
 public:
  /**
   * Attribute name
   */
  string attrName;

```

```
  /**
   * Attribute type
   */
  DataType attrType;

  /**
   * The max size of the attribute
   * If the attribute is CHAR(5), maxSize = 5
   */
  int maxSize;

  /**
   * Is the attribute not allowed to be null?
   */
  bool isNotNull;

  /**
   * Is the attribute required to be unique?
   */
  bool isUnique;

  /**
   * Constructor
   */
  Attribute(const string& attrName,
            const DataType& attrType,
            int maxSize,
            bool isNotNull = false,
            bool isUnique = false);

  /**
   * Destructor
   */
  ~Attribute();
};
```

- attrName: 属性名；
- attrType: 属性类型。例如，如果属性的类型为VARCHAR(5)，则attrType = VARCHAR；
- maxSize: 属性的最大长度。例如，如果属性的类型为VARCHAR(5)，则maxSize = 5；
- isNotNull: 如果属性不能为空，则为true；否则，为false；
- isUnique: 如果属性取值唯一，则为true；否则，为false；

注意，BadgerDB并不进行数据库完整性约束检查，因此本次实验只需对isNotNull和isUnique进行设置，而不进行检查。

2.1.3 TableSchema类

本实验定义了TableSchema类，用TableSchema类的对象来记录关系模式。TableSchema类的声明如下：

```
/**
 * Schema of Tables
 */
class TableSchema {
 private:
  /**
   * Table name
   */
  string tableName;

  /**
   * Attribute list
   */
```

```
14    vector<Attribute> attrs;
15
16    /**
17     * Is temporary table?
18     */
19    bool isTemp;
20
21  public:
22    /**
23     * Constructor
24     */
25    TableSchema(const string& tableName, bool isTemp = false);
26
27    /**
28     * Constructor
29     */
30    TableSchema(const string& tableName,
31                const vector<Attribute>& attrs,
32                bool isTemp = false);
33
34    /**
35     * Copy constructor
36     */
37    TableSchema(const TableSchema& tableSchema);
38
39    /**
40     * Destructor
41     */
42    ~TableSchema();
43
44    /**
45     * Create table schema from an SQL statement
46     */
47    static TableSchema fromSQLStatement(const string& sql);
48
49    /**
50     * Is the table temporary?
51     */
52    bool isTempTable() const;
53
54    /**
55     * Get table name
56     */
57    const string& getTableName() const;
58
59    /**
60     * Get the number of attributes
61     */
62    int getAttrCount() const;
63
64    /**
65     * Get the name of the num-th attribute
66     */
67    const string& getAttrName(int num) const;
68
69    /**
70     * Get the type of the num-th attribute
71     */
72    const DataType& getAttrType(int num) const;
73
74    /**
75     * Get the max size of the num-th attribute
76     */
77    int getAttrMaxSize(int num) const;
78
79    /**
80     * Is the num-th attribute not allowed to be null?
81     */
```

```
  bool isAttrNotNull(int num) const;

  /**
   * Is the num-th attribute required to be unique?
   */
  bool isAttrUnique(int num) const;

  /**
   * Set the type of the num-th attribute
   */
  void setAttrType(int num, const DataType& type);

  /**
   * Get the number of attribute by its name
   */
  int getAttrNum(const string& attrName) const;

  /**
   * Does the table contains the attribute?
   */
  bool hasAttr(const string& attrName) const;

  /**
   * Add an attribute to the table
   */
  void addAttr(const Attribute& attr);

  /**
   * Delete the num-th attribute
   */
  void deleteAttr(int num);

  /**
   * Print the schema
   */
  void print() const;
};
```

TableSchema类的属性如下：

- tableName: 表名；
- attrs: 全部属性的定义(每个属性的定义用Attribute类的对象来记录)；
- isTemp: 是否为临时表；

TableSchema类的每个方法的功能见代码注释。我们已经给出了TableSchema类全部方法的实现。

2.2 系统目录(System Catalog)

BadgerDB在系统目录(system catalog)中存储一个数据库的所有关系模式的定义。在BadgerDB中，每个表有一个编号(table Id)，编号的类型声明如下：

```
/**
 * Table Id
 */
typedef std::uint32_t TableId;
```

本实验定义了Catalog类，并用Catalog类的对象来存储数据库的所有关系模式。Catalog类的声明在catalog.h文件中，具体内容如下：

```
/**
 * System catalog
 */
class Catalog {
 private:
  /**
```

```
    * Database name
    */
   string dbName;

   /**
    * Mapping table name to table Id
    */
   map<string, TableId> tableIds;

   /**
    * Mapping table Id to table schema
    */
   map<TableId, TableSchema> tableSchemas;

   /**
    * Mapping table id to table filename
    */
   map<TableId, string> tableFilenames;

   /**
    * Next available table Id
    */
   TableId nextTableId;

 public:
   /**
    * Constructor
    */
   Catalog(const string& dbName);

   /**
    * Destructor
    */
   ~Catalog();

   /**
    * Get database name
    */
   const string& getDatabaseName() const;

   /**
    * Get table Id
    */
   const TableId& getTableId(const string& tableName) const;

   /**
    * Get table schema
    */
   const TableSchema& getTableSchema(const TableId& id) const;

   /**
    * Get table file
    */
   const string& getTableFilename(const TableId& id) const;

   /**
    * CREATE TABLE
    */
   TableId addTableSchema(const TableSchema& tableSchema,
                          const string& tableFilename);

   /**
    * DROP TABLE
    */
   void deleteTableSchema(const TableId& id);

   /**
    * ALTER TABLE
```

```
    */
  void setTableSchema(const TableId& id, const TableSchema& tableSchema);
};
```

Catalog类的属性如下：

- dbName: 数据库名；
- tableIds: 表名到表号的映；
- tableSchemas: 表号到表的模式定义的映射；
- tableFilenames: 表号到表的数据文件名的映射；
- nextTableId: 下一个可用的表号，nextTableId单调递增。

Catalog类的方法的功能见代码注释。本实验已经给出了Catalog类所有方法的实现。

2.3 存储管理器(Storage Manager)

存储管理器(storage manager)的功能是向表中插入或删除元组。在BadgerDB中，我们使用堆文件(heap file)存储表，并提供了HeapFileManager类，用于实现在表中插入和删除元组的功能。HeapFileManager类的声明如下：

```
/**
 * Heap file manager for inserting and deleting tuples
 */
class HeapFileManager {
 public:
  /**
   * Insert a tuple to a table
   */
  static RecordId insertTuple(const string& tuple, File& file, BufMgr* bufMgr);

  /**
   * Delete a tuple from a table
   */
  static void deleteTuple(const RecordId& rid, File& file, BufMgr* bugMgr);

  /**
   * Create a tuple from an SQL statement
   */
  static string createTupleFromSQLStatement(const string& sql,
                                            const Catalog* catalog);
};
```

HeapFileManager类声明了3个静态方法。

- insertTuple: 向关系中插入一条元组。该方法有3个参数：
  - tuple: 元组的值。我们用string类型的值来存储元组，但这里tuple的内容并非字符串，而是字节序列(sequence of bytes)。请阅读insertTuple的实现代码，了解元组的内部表示(tuple layout)方法，以便对tuple的内容进行读写。注意，tuple在数据库内部的表示和存储形式采用课上讲过的tuple layout方法(但是没有头部)，而不是将元组的SQL字符串表示存入tuple中。
  - file: 关系的数据文件的句柄。
  - bufMgr: 指向缓冲区管理器对象的指针。

  按照堆文件组织方式插入新元组tuple。如果元组插入成功，返回该元组的记录号(RecordId类型)。
- deleteTuple: 从关系中删除一条元组。该方法有3个参数，后2个参数file和bufMgr与insertTuple方法的同名参数相同。deleteTuple方法的rid参数是待删除的元组的记录号(RecordId类型)。
- createTupleFromSQLStatement: 该方法根据输入的INSERT语句和插入关系的模式，创建一条元组，并将该元组返回。

2.4 查询执行器(Query Executor)

本实验要求编写自然连接执行器(natural join operation executor)来实现自然连接(natural join)操作，对两个关系进行自然连接，具体实现基于块的嵌套循环连接(Block-based Nested Loop Join)。感兴趣的同学还可以实现一趟连接算法(One-Pass Join)和Grace哈希连接算法(Grace Hash Join)。

本实验定义了`JoinOperator`类作为各种连接操作执行器的基类(base class)。`JoinOperator`类的声明如下：

```
/**
 * Join Operator
 */
class JoinOperator {
 protected:
  /**
   * Data file of the left table
   */
  const File& leftTableFile;

  /**
   * Data file of the right table
   */
  const File& rightTableFile;

  /**
   * Schema of the left table
   */
  const TableSchema& leftTableSchema;

  /**
   * Schema of the right table
   */
  const TableSchema& rightTableSchema;

  /**
   * Schema of the result table
   */
  TableSchema resultTableSchema;

  /**
   * System catalog
   */
  const Catalog* catalog;

  /**
   * Buffer pool manager
   */
  BufMgr* bufMgr;

  /**
   * Is the executor completed
   */
  bool isComplete;

  /**
   * Number of result tuples
   */
  int numResultTuples;

  /**
   * Number of buffer pages actually used by the executor
   */
  int numUsedBufPages;

  /**
   * Number of I/Os carried out by the executor
   */
  int numIOs;
```

```
60
61  public:
62   /**
63    * Constructor
64    */
65   JoinOperator(const File& leftTableFile,
66                const File& rightTableFile,
67                const TableSchema& leftTableSchema,
68                const TableSchema& rightTableSchema,
69                const Catalog* catalog,
70                BufMgr* bufMgr);
71
72   /**
73    * Destructor
74    */
75   ~JoinOperator();
76
77   /**
78    * Is the algorithm complete?
79    */
80   bool isCompleted() const;
81
82   /**
83    * Get the operator's name
84    */
85   virtual string getOperatorName() const;
86
87   /**
88    * Print the running statistics of the executor
89    */
90   virtual void printRunningStats() const;
91
92   /**
93    * Execute the join algorithm
94    * If succeeded, return true
95    */
96   virtual bool execute(int numAvailableBufPages, File& resultFile) = 0;
97
98   /**
99    * Get the schema of the result table
100   */
101  const TableSchema& getResultTableSchema() const;
102
103  /**
104   * Get number of result tuples
105   */
106  int getNumResultTuples() const;
107
108  /**
109   * Get number of buffer pages used by the executor
110   */
111  int getNumUsedBufPages() const;
112
113  /**
114   * Get number of I/Os carried out by the executor
115   */
116  int getNumIOs() const;
117
118  /**
119   * Create the result schema using the input schemas
120   */
121  static TableSchema createResultTableSchema(
122      const TableSchema& leftTableSchema,
123      const TableSchema& rightTableSchema);
124
125 protected:
126  /**
127   * Get common attributes in all input tables
```

```
    */
  vector<Attribute> getCommonAttributes(
      const TableSchema& leftTableSchema,
      const TableSchema& rightTableSchema) const;

  /**
   * Join two tuples
   */
  string joinTuples(string leftTuple,
                    string rightTuple,
                    const TableSchema& leftTableSchema,
                    const TableSchema& rightTableSchema) const;
};
```

JoinOperator类的属性如下：

- leftTableFile: 左关系文件的句柄；
- rightTableFile: 右关系文件的句柄；
- leftTableSchema: 左关系的模式；
- rightTableSchema: 右关系的模式；
- resultTableSchema: 结果关系的模式；
- catalog: 系统目录对象的指针；
- bufMgr: 缓冲池管理器对象的指针；
- isComplete: 算法执行是否结束；
- numResultTuples: 结果元组数量；
- numUsedBufPages: 算法执行过程中实际使用的缓冲页面数；
- numIOs: 算法执行过程中实际执行的I/O数；

JoinOperator类的方法的功能见代码注释。你需要实现其中1个方法：

- execute: 该方法为纯虚拟函数(pure virtual method)，需要在JoinOperator类的子类中进行实现。注意：在实现算法的时候，不仅要编程实现连接操作，还要记录结果元组数(numResultTuples)、算法执行过程中使用的缓冲页面数(numUsedBufPages)和I/O次数(numIOs)。

为了实现基于块的嵌套连接算法，本实验声明了NestedLoopJoinOperator类。该类继承了JoinOperator类。你需要实现NestedLoopJoinOperator类的execute方法。

为了实现一趟连接算法，本实验声明了OnePassJoinOperator类。该类继承了JoinOperator类。你需要实现OnePassJoinOperator类的execute方法。

为了实现Grace哈希连接算法，本实验声明了GraceHashJoinOperator类。该类继承了JoinOperator类。你需要实现GraceHashJoinOperator类的execute方法。

本实验只要求实现NestedLoopJoinOperator类的execute方法。

如果你感兴趣，还可以实现一下OnePassJoinOperator类的execute方法和GraceHashJoinOperator类的execute方法。

2.5 表扫描器(Table Scanner)

本实验定义了TableScanner类，用于对表进行扫描和打印。TableScanner类的声明如下：

```
/**
 * Table scanner
 */
class TableScanner {
 private:
  /**
```

```
    * Table filename
    */
   const File& tableFile;

   /**
    * Table schema
    */
   const TableSchema& tableSchema;

   /**
    * Buffer pool manager
    */
   BufMgr* bufMgr;

 public:
   TableScanner(const File& tableFile,
                const TableSchema& tableSchema,
                BufMgr* bufMgr);

   ~TableScanner();

   /**
    * Print tuples in the table
    */
   void print() const;
};
```

我们已经实现了TableScanner::print()函数。

2.6 测试代码

我们在main.cpp文件中给出了详细的测试代码及注释。你可以对缓冲区页面数、表的元组数、连接算法可以使用的缓冲区页数进行调整。

3 实验内容

1. 阅读代码。本次实验需要重点阅读的代码如下：
   - TableSchema类的声明(了解如何获取一个关系中的属性)；
   - HeapFileManager类的createTupleFromSQLStatement方法(了解元组的内部表示方法)；
   - HeapFileManager的insertTuple方法(了解元组在文件中如何存储)；
   - JoinOperator中的方法。
2. 将实验3中正确实现的buffer.cpp文件复制到src目录下，并实现NestedLoopJoinOperator类的execute方法，完成BadgerDB的查询执行功能。本次实验的代码量约为100–200行。
3. 本实验在main.cpp的main函数中提供了本次实验测试过程的代码，请按照此代码给出的过程进行测试。

4 实验要求

1. 本实验由每名学生独立完成。
2. 本实验提供了项目的C++源代码。你的代码风格应当符合C++面向对象程序设计的代码风格，包含定义明确的类及清晰的接口。使用C语言的编程风格将被扣分。
3. 代码须包含Doxygen风格的注释。
4. 使用编译选项-Wall打开所有编译警告。实验源代码的Makefile文件中默认使用了-Wall编译选项。

5. 善于使用工具。例如，使用make编译项目；使用makedepend自动生成依赖；使用perl、python或bash编写测试脚本；使用valgrind捕捉内存错误；使用gdb进行调试；使用git进行版本控制。

6. 实验报告以PDF文件提交，文件命名规则为“实验4-学号-姓名-报告.pdf”；提交源代码时只提交源文件的压缩包（不包含二进制文件），文件命名规则为“实验4-学号-姓名-源代码.zip”。

7. 在进行代码检查时，老师会使用新的测试用例，重新编译你的源代码，并进行测试。因此，请不要修改现有代码的接口。如果你修改了接口并因此导致程序无法编译，则你会被扣分。

5 实验评价

本实验的成绩构成如下:

- 程序正确性: 60%
- 代码风格: 10%
- 代码讲解: 10%
- 实验报告: 20%