实验3: 查询执行器(2020春)

主讲教师：邹兆年(znzou@hit.edu.cn)

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

本次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)

为了表示和存储BadgerDB的关系模式，我们在schema.h和schema.cpp中定义了相关的数据类型和类。

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

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 };
```

同学可以自行增加对DATETIME、DECIMAL、REAL等数据类型支持。

Atrribute类

本实验定义了Attribute类，用Attribute类的对象记录关系的属性(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，如果属性为VARCHAR(5)类型，则attrType = VARCHAR；
- maxSize: 属性的最大长度。例如，如果属性为VARCHAR(5)类型，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，但不进行检查。

TableSchema类

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

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

  /**
   * Attribute list
   */
  vector<Attribute> attrs;

  /**
   * Is temporary table?
   */
  bool isTemp;

 public:
  /**
   * Constructor
   */
  TableSchema(const string& tableName, bool isTemp = false);

  /**
   * Constructor
   */
  TableSchema(const string& tableName,
              const vector<Attribute>& attrs,
              bool isTemp = false);

  /**
   * Copy constructor
```

```
     */
    TableSchema(const TableSchema& tableSchema);

    /**
     * Destructor
     */
    ~TableSchema();

    /**
     * Create table schema from an SQL statement
     */
    static TableSchema fromSQLStatement(const string& sql);

    /**
     * Is the table temporary?
     */
    bool isTempTable() const;

    /**
     * Get table name
     */
    const string& getTableName() const;

    /**
     * Get the number of attributes
     */
    int getAttrCount() const;

    /**
     * Get the name of the num-th attribute
     */
    const string& getAttrName(int num) const;

    /**
     * Get the type of the num-th attribute
     */
    const DataType& getAttrType(int num) const;

    /**
     * Get the max size of the num-th attribute
     */
    int getAttrMaxSize(int num) const;

    /**
     * Is the num-th attribute not allowed to be null?
     */
    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类每个方法的功能见代码注释。除其中2个方法外，我们已经给出全部其他方法的实现。你需要实现下面2个方法：

- fromSQLStatement：根据输入字符串参数sql，创建TableSchema类的对象，并返回该对象的指针。输入参数sql为完整的SQL CREATE TABLE语句字符串。CREATE TABLE语句中属性类型只能是INT、CHAR或VARCHAR，属性完整性约束只能为NOT NULL或UNIQUE。

```
CREATE TABLE r (
  id INT NOT NULL UNIQUE,
  val VARCHAR(5)
);
```

- print方法：打印关系模式的定义，具体打印格式由你自己定义，可以参考MySQL、PostgresSQL等DBMS的模式定义输出格式。

2.2 系统目录(System Catalog)

BadgerDB在系统目录(catalog)中存储一个数据库的所有关系的模式定义。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模式信息。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: 下一个可用的表的编号，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)的形式存储表，并提供了HeapFileManager类，用于实现对向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);
};
```

Catalog类声明了3个静态方法。这3个方法都需要你自己实现。

- 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字符串来存储元组，但这里tuple的内容并非字符串，而是字节序列。你需要自己设计元组的内部表示(tuple layout)方法，并编程实现，对tuple的内容进行读写。
  - file: 关系的数据文件的句柄。
  - bufMgr: 指向缓冲池管理器对象的指针。

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SQL 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，具体实现一趟连接算法(One-Pass Join)、基于块的嵌套循环连接(Block-based Nested Loop Join)、Grace哈希连接算法(Grace Hash Join)。其中，Grace哈希连接算法实现为选作。

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;

  public:
    /**
     * Constructor
     */
    JoinOperator(const File& leftTableFile,
                 const File& rightTableFile,
                 const TableSchema& leftTableSchema,
                 const TableSchema& rightTableSchema,
                 const Catalog* catalog,
                 BufMgr* bufMgr);

    /**
     * Destructor
     */
    ~JoinOperator();

    /**
     * Is the algorithm complete?
     */
    bool isCompleted() const;
```

```
    /**
     * Get the operator's name
     */
    virtual string getOperatorName() const;

    /**
     * Print the running statistics of the executor
     */
    virtual void printRunningStats() const;

    /**
     * Execute the join algorithm
     * @return If succeeded, return true
     */
    virtual bool execute(int numAvailableBufPages, File& resultFile) = 0;

    /**
     * Get the schema of the result table
     */
    const TableSchema& getResultTableSchema() const;

    /**
     * Get number of result tuples
     */
    int getNumResultTuples() const;

    /**
     * Get number of buffer pages used by the executor
     */
    int getNumUsedBufPages() const;

    /**
     * Get number of I/Os carried out by the executor
     */
    int getNumIOs() const;

    /**
     * Create the result schema using the input schemas
     */
    static TableSchema createResultTableSchema(
        const TableSchema& leftTableSchema,
        const TableSchema& rightTableSchema);
};
```

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；
- numbs: 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。同学们需要实现其中2个方法：

- createResultTableSchema: 根据leftTableSchema和rightTableSchema来创建结果关系的模式。注意，这里进行的是自然连接。
- 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)。

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

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

如果你感兴趣，还可以实现一下GraceHashJoinOperator类，实现Grace哈希连接算法。

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;
};
```

你需要实现TableScanner::print()函数，实现打印表中元组的功能。具体打印格式由你自己定义，可以参考MySQL、PostgresSQL等DBMS的元组输出格式。

2.6 测试代码

本次实验的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

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);
```

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

We have defined this project so that you can understand and reap the full benefits of object-oriented programming using C++. Your coding style should continue this by having well-defined classes and clean interfaces. Reverting to the C (low-level procedural) style of programming is not recommended and will be penalized. The code should be well-documented, using Doxygen style comments. Each file should start with your name and student id, and should explain the purpose of the file. Each function should be preceded by a few lines of comments describing the function and explaining the input and output parameters and return values.

5. 实验提交

源代码

You are required to submit all the necessary material in a single zipped folder (use GZip or WinZip). Your folder should be named **实验3-源代码-姓名-学号** and include only the source code files (no binaries). We will compile your implementation, link it with our test driver, and run tests. Since we are supposed to be able to test your code with any valid driver, it is important to be faithful to the exact definitions of the interfaces as specified here. If you alter these interfaces and your code does not compile, you will be penalized.

实验报告

本次实验要求撰写实验报告。在实验报告中介绍8个方法的程序设计和实现方法。实验报告的文件名为**实验3-报告-姓名-学号**。