CS44800 Project 3
Relational Operators and Lazy Query Evaluation Pipelines
Spring 2016

Due: Monday 28 March, 2016, 11:59PM
(There will be a 10% penalty for each late day up to four days. The assignment will not be accepted afterwards.)

Notes:

- This project should be carried out in groups of two. Choose a partner as soon as possible. Each group will take the same grade, so make sure that every member backs up his teammate (no excuses will be accepted).

Part 1: Scan Operators - From Records to Tuples

As you have learned in class, a typical database query processor breaks down queries into trees of relational operators, implemented as iterators. The iterators (e.g., *HeapScan* and *HashScan*) deal with file access directly, and return records or their ids. In this project, you will build and use a higher-level view of these records with the provided classes *Schema* and *Tuple*. Each high-level relational operator you will implement inherits the abstract class *Iterator*, which contains a schema and requires the following methods:

```
public Schema schema
public void restart()
public boolean isOpen()
public void close()

public boolean hasNext()
public Tuple getNext()
```

Your task is to implement the following wrappers for the heap and index scans:

1. **FileScan**
   A *HeapScan* that returns *Tuples* instead of byte[] 's
2. **KeyScan**
   A *HashScan* that returns *Tuples* instead of *RID*s
3. **IndexScan**
   A *BucketScan* that returns *Tuples* instead of *RID*s

Some useful hints and tips:

- Each constructor should initialize the inherited field *schema*. (i.e. it's given as a parameter to these three iterators)
- Don't be surprised by how little code these classes require
- *HashScan* scans the hash index for records having a given search key. *BucketScan* scans the whole hash index. Those classes only return RIDs. You have to build wrapper classes KeyScan and IndexScan that return Tuple.

PART 2: Primitive Operators

Now that you have the basic leaf nodes of most query trees, you can make some more interesting iterators. Your next task is to implement the three fundamental operations of relational algebra:

1. **Selection**
   Filters another iterator on a set of *Predicate*s. When there are multiple predicates, they are connected by operator "AND" by default (i.e. simply call *evaluate*() on each one)
2. **Projection**
   Removes (projects) columns from another iterator. Note: duplicates are OK.
3. **Join**
   The code for "Nested Loops Join" is provided for you in SimpleJoin.java. You are required to implement a **Hash-Join**. You may consider to study the code in SimpleJoin.java and figure out how you can extend it to support Hash Join. Please refer to the textbook before implementing the hash-join.

PART 3: Query Evaluation Pipelines

Now that you have a set of operators, you are ready to use these operators to form query evaluation pipelines (QEPs, for short) that can evaluate some queries.
To simplify your task, you are given the code for some simple QEPs that correspond to some simple queries. You will find that code in ROTest.java. You have to study that code in order to understand how a schema can be created, data can be inserted, and operators can be connected in Minibase.

Consider the relational schema of two tables:

- **Employee (EmpId, Name, Age, Salary, DeptID), and**
- **Department (DeptId, Name, MinSalary, MaxSalary)**

For simplicity, assume one-to-one relationship between the two tables.

You are required to programmatically form a QEP for each of the queries given below:

1. Display for each employee his ID, Name and Age
2. Display the Name for the departments with MinSalary = MaxSalary
3. For each employee, display his Name and the Name of his department as well as the maximum salary of his department
4. Display the Name for each employee whose Salary is greater than the maximum salary of his department.

Note that whenever you need to connect a join operator to your QEP, you can use either the hash-join operator you implemented, or the nested-loops join operator that is already given to you. However, your final submission of Part 3 should use the hash-join operator.

Following the paradigm in ROTest.java, you have to write **one test for each of the above queries** inside QEPTest.java.
Your main() method should take one argument that corresponds to the path of the folder in which the data for the Employee and Department tables exists. A sample folder is given for you, where there are two files named Employee.txt and Department.txt. The data in these files is in comma-separated format. The first line in each of these files corresponds to the schema, e.g., (Empid, Name, Age, Salary, DeptID). Your program should read the data of each of these files starting from the second line.

Getting Started

Skeleton of the code is available here, and the documentation is available here.
Note that this code skeleton is a complete starting point for the project, i.e., the *bufmgr*, *heap*, and *index* packages are provided for you (in jar files).

To test your code for Parts 1 and 2, simply run the provided *ROTest.java* test driver.

To implement Part 3, you need to write test cases in ***QEPTest.java*** by mimicking ROTest.java. Unlike ROTest.java, where the data is hardwired in the code, in QEPTest.java, you need to read the files Employee.txt and Department.txt before loading the data into Minibase.

Note
Implementing the hash-join operator may consume more time than the other operators. Because the code for nested-loops join is already given to you, you can start Part 3 once you have implemented the selection and projection operators. At this point in time, one way to distribute the work between two partners is to have one of them working on the hash-join operator, and the other working on Part 3. This is just a recommendation, and it is totally up to you on how to distribute the work between you as partners.

Turnin

You should turn in your code with the Makefile and a readme file. All files need to be zipped in a file named: **your_career_login1_your_career_login2_ro.zip**.
In the readme file, put anything you would like us to know. We should be able to compile/run your program using make on a CS department Unix machine.
Do not change the directory structure of the code. The directory structure of your zip file should be identical to the directory structure of the provided zip file (i.e., having the directory src, the Makefile, ...), except the top-level name (should be your career login above). Your grade may be deduced 5% off if you don't follow this.