CS54100 - Programming Assignment 3 Relational Operators and Lazy Query Evaluation Pipelines

- Due: 11:59PM, Wednesday, November 16, 2016. Submit using Blackboard.
- (There will be a 10% penalty for each late day. After 5 late days, the project will not be accepted.)

Part 1: Scan Operators - From Records to Tuples

As you have learned in class, a typical database query processor optimizer breaks down queries into trees of relational operators, implemented as iterators. The iterators (i.e., *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:

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

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

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

1. **FileScan**
2. A *HeapScan* that returns *Tuples* instead of byte[] 's
3. **KeyScan**
4. A *HashScan* that returns *Tuples* instead of *RID*s
5. **IndexScan**
6. 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*.
- You are provided with code for *SimpleJoin*, which performs a simple nested loop join. It can be useful to study *SimpleJoin* to understand the semantics of the Iterator interface.

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 within a Selection operator, 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 **HashJoin**. 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.

HashJoin

As described in the course textbook and slides, the Hash-Join algorithm consists of two phases, the partitioning and the probing phase. You are free to do any implementation of *HashJoin* you wish as long as the algorithm is followed correctly. However, the following are some hints to get started:

You do not have to implement partitioning entirely from scratch, as the *BucketScan* operator (specifically, the *IndexScan* wrapper implemented in Part 1) can be used to accomplish this.

Although an *IndexScan* can be used in order to perform the partitioning, the input nodes to a *HashJoin* can be any Iterator. You need to construct an *IndexScan* out of the input nodes, but you need to avoid unnecessary I/Os while doing so. You need to consider the following cases:

- If an input node is already an *IndexScan*, do not rebuild the index. Use it as-is.

- If an input node is a *FileScan*, you will have to construct an *IndexScan*. Create a temporary *HashIndex* and make use of the existing file in the *FileScan* in order build the *IndexScan*. You may add additional methods to the *FileScan* beyond those specified in the public interface in order to facilitate this.
- For all other input nodes, you will have to materialize the result into temporary files/indexes in order to construct an *IndexScan*.

For the probing phase, you need a good in-memory hash structure that can handle duplicate keys. The *HashTableDup* class, provided in the skeleton code, is an extension of Java's *HashTable* that supports this. You can use this to help you implement the probing phase as well.

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. Note that these files are provided for your convenience for testing, and the contents will not be the same as what you are graded on. You are encouraged to insert your own data into these files in order to test corner cases.

Getting Started

Skeleton of the code is available on blackboard, 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 in order to load the data into Minibase. Loading the data should be done once at the beginning, not once per test. The grading will be performed by running all tests in *QEPTest* at once, not individually, so loading the data multiple times will result in incorrect answers.

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.