CSI 508. Database Systems I – Fall 2017

Programming Assignment II

The total grade for this assignment is 100 points. The deadline for this assignment is **11:59 PM, November 27, 2017**. *Submissions after the deadline will not be accepted.* Students are required to enter the UAlbany Blackboard system and then upload a compressed file (with the names of the students as the file name in the form of [first_name1]_[last_name1]__[first_name2]_[last_name2]) that contains the eclipse project directory and a short document that describes:

- the methods implemented (if implemented by a team of two students, list the names of the students for each method)
- any missing or incomplete elements of the code
- any changes made to the original API
- the amount of time spent for this assignment
- suggestions or comments if any

In this programming assignment, you need to implement a set of operators for `SimpleDB`. Once these operators are implemented, `SimpleDB` can execute queries over tables.

In programming assignment I, we ignored the buffer pool management problem (i.e., one that arises when we reference more pages than we can fit in memory). In this assignment, you will design an eviction policy to flush stale pages from the buffer pool. This assignment requires you to run Eclipse on your machine (refer to Appendix A of Programming Assignment I). For importing the "simpledb-2" project and copying your previous code to this project, see Appendix A of this document. Regarding test suites and the overall architecture of `SimpleDB`, refer to Appendices B and D of Programming Assignment I, respectively. Documents on `SimpleDB` API can also be generated automatically using `javadoc` (refer to Appendix C of Programming Assignment I for more information). As a side note, the unit tests provided for this assignment are to guide your implementation, but are not intended to be comprehensive or perfect. This assignment requires you to write a fair amount of code, so you should start early! If you find any bug or have suggestions, please contact the instructor (`jhh@cs.albany.edu`).

Part 1. Filter (20 points)

Recall that the `DbIterator` interface declares basic operations supported by relational algebraic operators. The `Filter` operator enables queries that are more interesting than a table scan. It returns only the tuples that satisfy a `Predicate` specified as part of its constructor. In other words, it filters out all tuples that do not match the `Predicate`.

In this part, you need to implement the skeleton methods in `Filter.java`. For this, it might be helpful to understand the implementation of other operators (e.g., `Project`). At this point, your code should pass the unit tests in `FilterTest`. Furthermore, your code should be able to pass the `FilterTest` system test.

Part 2. Join (20 points)

The `Join` operator joins tuples from its two children according to a `JoinPredicate` that is passed in as part of its constructor. We only require a simple nested loops join, but you may explore more interesting join implementations. Describe your implementation in your writeup.

In this part, you need to implement the skeleton methods in `Join.java`. At this point, your code should pass the unit tests in `JoinTest`. Furthermore, your code should be able to pass the `JoinTest` system test.

Part 3. Aggregates (20 points)

The `Aggregate` operator supports basic SQL aggregates with a `GROUP BY` clause. These aggregates are `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. The `AVG` aggregate needs to use integer division since `SimpleDB` only supports integers. The `Aggregate` operator supports aggregates over a single field, and grouping by a single field. In order to calculate aggregates, the `Aggregate` operator uses either the `IntAggregator` or `StringAggregator` depending on the type of the aggregation field. These `IntAggregator` and `StringAggregator` implement the `Aggregator` interface.

When an `Aggregator` is constructed, it is told what aggregation operation it needs to apply. For every tuple obtained from the child iterator of the `Aggregate` (see the `open()` method in `Aggregate`), the `merge(Tuple tup)` method of the `Aggregator` needs to be called. Given a tuple, this `merge(Tuple tup)` method merges that tuple into the existing calculation of an aggregate (e.g., in the case of `COUNT`, it increments a variable representing the count of tuples). After the `merge(Tuple tup)` method is used for all of the relevant tuples, the `iterator()` method of the `Aggregator` needs to be called (see the `open()` method in `Aggregate`). This method needs to return a `DbIterator` of aggregation results. Each tuple in the result is a pair of the form (`groupValue`, `aggregateValue`), unless the value of the group by field was `Aggregator.NO_GROUPING`, in which case the result is a single tuple of the form (`aggregateValue`). Whenever `next()` is called on the `DbIterator` mentioned above, an aggregate result (i.e., an aggregate value of a group) needs to be returned.

The aforementioned implementation requires space linear in the number of distinct groups. For the purposes of this assignment, you do not need to worry about the situation where the number of groups exceeds available memory.

In this part, you need to implement the skeleton methods in `IntAggregator.java`, `StringAggregator.java`, `Aggregator.java`, and `Aggregate.java`. `StringAggegator` only needs to support the `COUNT` aggregate since the other aggregates do not make sense for strings. At this point, your code should pass the unit tests `IntAggregatorTest`, `StringAggregatorTest`, and `AggregateTest`. Furthermore, the code should pass the `AggregateTest` system test.

Part 4. HeapFile Mutability (20 points)

Now, we will begin to implement methods to support queries that remove tuples from tables. To delete a tuple, you first need to obtain the `RecordID` of the tuple (by using `getRecordId()`), which allows you to find the page containing the tuple. This part requires a correct implementation of the `deleteTuple(Tuple t)` method in `HeapPage.java` (see Programming Assignment I).

In this part, you need to complete the `deleteTuple(TransactionId tid, Tuple t)` method of the `HeapFile` class. This method needs to access the page containing tuple `t` using the `getPage(TransactionId tid, PageId pid, Permissions perm)` method of the `BufferPool`.

In this part, you also need to implement the `deleteTuple()` method in `BufferPool.java`.

This method needs to call the `deleteTuple(TransactionId tid, Tuple t)` method on the `HeapFile` associated with the table being modified. The table can be found from tuple `t` and the `HeapFile` can be found from the system catalog (refer to the `getPage(TransactionId tid, PageId pid, Permissions perm)` method). The above extra level of indirection (i.e., trying to delete a tuple via the BufferPool and then a HeapFile) is needed to support other types of files (e.g., indices).

Unit tests for HeapFile.deleteTuple() and BufferPool are not provided.

Part 5. Deletion (20 points)

Now you have written all of the HeapFile machinery to remove tuples, so you can implement the Delete operator. In a query plan that deletes tuples from a table, the top-most operator is a special Delete operator that modifies the pages on disk. The Delete operator implements DbIterator, accepting a stream of tuples to delete. This operator returns the number of affected tuples. This is implemented by returning a single tuple with one integer field, containing the count.

The Delete operator deletes the tuples it reads from its child operator specified in its constructor. To delete a tuple, it needs to use the deleteTuple() method of the buffer pool.

In this part, you need to implement the skeleton methods in Delete.java. At this point, your code should pass the DeleteTest system test.

Final Remark

You should be able to pass all of the tests in the ant systemtest target, which is the goal of this assignment.

The Java code below implements a simple join query between two tables, each consisting of three columns of integers (some_data_file1.dat and some_data_file2.dat are binary representation of the pages from this file). This code is equivalent to the following SQL statement:

```
SELECT *
  FROM some_data_file1, some_data_file2
  WHERE some_data_file1.field1 = some_data_file2.field1
  AND some_data_file1.id > 1
```

```
import java.io.*;

public class jointest {

    public static void main(String[] argv) {
        // construct a 3-column table schema
        Type types[] = new Type[]{ Type.INT_TYPE, Type.INT_TYPE, Type.INT_TYPE };
        String names[] = new String[]{ "field0", "field1", "field2" };

        TupleDesc td = new TupleDesc(types, names);

        // create the tables, associate them with the data files
        // and tell the catalog about the schema  the tables.
        HeapFile table1 = new HeapFile(new File("some_data_file1.dat"), td);
        Database.getCatalog().addTable(table1, "t1");

        HeapFile table2 = new HeapFile(new File("some_data_file2.dat"), td);
        Database.getCatalog().addTable(table2, "t2");

        // construct the query: we use two SeqScans, which spoonfeed
        // tuples via iterators into join
        TransactionId tid = new TransactionId();

        SeqScan ss1 = new SeqScan(tid, table1.getId(), "t1");
        SeqScan ss2 = new SeqScan(tid, table2.getId(), "t2");

        // create a filter for the where condition
        Filter sf1 = new Filter(new Predicate(0,
                                Predicate.Op.GREATER_THAN, new IntField(1)),  ss1);
```

```
        JoinPredicate p = new JoinPredicate(1, Predicate.Op.EQUALS, 1);
        Join j = new Join(p, sf1, ss2);

        // and run it
        try {
            j.open();
            while (j.hasNext()) {
                Tuple tup = j.next();
                System.out.println(tup);
            }
            j.close();
            Database.getBufferPool().transactionComplete(tid);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

In the above code, both tables have three integer fields. To express this, we create a `TupleDesc` object and pass it an array of `Type` objects indicating field types and `String` objects indicating field names. Once we have created this `TupleDesc`, we initialize two `HeapFile` objects representing the tables. Once we have created the tables, we add them to the `Catalog` (if this were a database server that was already running, we would have this catalog information loaded; we need to load this only for the purposes of this test).

After the database system is initialized, we create a query plan. Our plan consists of two `SeqScan` operators that scan the tuples from each file on disk, connected to a `Filter` operator on the first `HeapFile`, connected to a `Join` operator that joins the tuples in the tables according to the `JoinPredicate`. In general, these operators are instantiated with references to the appropriate table (in the case of `SeqScan`) or child operator (in the case of `Join`). The test program then repeatedly calls `next()` on the `Join` operator, which in turn pulls tuples from its children. As tuples are output from the `Join`, they are printed out on the command line.

Appendix A. Project Preparation

1. Run Eclipse. In the menu bar, choose "File" and then "Import". Next, select "General" and "Existing Projects into Workspace". Then, click the "Browse" button and select the "`2017f_p2.zip`" file contained in this assignment package.

2. In the `src/simpledb` package of the `simpledb-1` project, select *only* the Java files to copy. Copy the files (e.g., right click and then choose the "Copy" menu item, or press [Ctrl] and "C" at the same time) and then paste them into the `src/simpledb` package of the `simple-2` project. If "Confirm Overwriting" dialog box shows up, click the "Yes" button to overwrite files.