Lab 8 (PA 3)

Programming Assignment 3

Released: 10/30

Due: 11/13

In this assignment, you will write a set of operators for SimpleDB to implement table modifications (e.g., insert and delete records), selections, joins, and aggregates. These will build on top of the foundation that you wrote in PA 1 to provide you with a database system that can perform simple queries over multiple tables.

Additionally, you will also utilize the buffer eviction code you have developed in previous assignment. You do not need to implement transactions or locking now.

The remainder of this document gives some suggestions about how to start coding, describes a set of exercises to help you work through the assignment, and discusses how to hand in your code. This assignment requires you to write a fair amount of code, since you have a midterm to take, we encourage you to start early!

1. Getting started

1.1. Adding skeleton code for assignment 3

You should begin with the code you submitted for PA 1 and 2 (if you did not submit code for PA 1 and PA 2, or your solution didn't work properly, contact us to discuss options). We have provided you with extra test cases for this assignment that are not in the original code distribution you received. We reiterate that the unit tests we provide are to help guide your implementation along, but they are not intended to be comprehensive or to establish correctness.

Y ill need to add these new test cases to your work. The easiest way to do this is to untar the new code in the same directory as your top-level simpledb directory, as follows:

- Make a backup of your previous solution.
- Change to the directory that contains your top-level simpledb code.
- Download the new tests and skeleton code for PA 3 from CS660-pa3.zip (Piazza).
- Extract the new files for PA 3 and merge with your previous solution.
- **Eclipse users** will have to take one more step for their code to compile. (First, make sure the project is in your workspace in Eclipse following similar steps as done in PA 1.) Under the package explorer, right click the project name, and select **Properties**. Choose **Java Build Path** on the left-hand-side, and click on the **Libraries** tab on the right-hand-side. Push the **Add JARs...** button, select **zql.jar** and **jline-0.9.94.jar**, and push **OK**, followed by **OK**. You may have already done this step from PA 2; in that case you can ignore this step . Your code should now compile.

1.2. Implementation hints

As before, we encourage you to read through this entire document to get a feel for the high-level design of SimpleDB before you write code.

We suggest exercises along this document to guide your implementation, but you may find that a different order makes more sense for you. As before, we will grade your assignment by looking at your code and verifying that you have passed the test for the ant targets `test` and `systemtest`. See Section 3.4 for a complete discussion of grading and list of the tests you will need to pass.

Here's a rough outline of one way you might proceed with your SimpleDB implementation; more details on the steps in this outline, including exercises, are given in Section 2 below.

- Implement the operators `Filter` and `Join` and verify that their corresponding tests work. The Javadoc comments for these operators contain details about how they should work. We have given you implementations of `Project` and `OrderBy` which may help you understand how other operators work.
- Implement `IntegerAggregator` and `StringAggregator`. Here, you will write the logic that actually computes an aggregate over a particular field across multiple groups in a sequence of input tuples. Use integer division for computing the average, since SimpleDB only supports integers. StringAggegator only needs to support the COUNT aggregate, since the other operations do not make sense for strings.
- Implement the `Aggregate` operator. As with other operators, aggregates implement the `DbIterator` interface so that they can be placed in SimpleDB query plans. Note that the output of an `Aggregate` operator is an aggregate value of an entire group for each call to `next()`, and that the aggregate constructor takes the aggregation and grouping fields.
- Implement the methods related to tuple insertion, deletion, and page eviction in `BufferPool`. You do not need to worry about transactions at this point.
- Implement the `Insert` and `Delete` operators. Like all operators, `Insert` and `Delete` implement `DbIterator`, accepting a stream of tuples to insert or delete and outputting a single tuple with an integer field that indicates the number of tuples inserted or deleted. These operators will need to call the appropriate methods in `BufferPool` that actually modify the pages on disk. Check that the tests for inserting and deleting

methods in BufferPool that actually modify the pages on disk. Check that the tests for inserting and deleting tuples work properly

- Note that SimpleDB does not implement any kind of consistency or integrity checking, so it is possible to insert duplicate records into a file and there is no way to enforce primary or foreign key constraints.

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

Finally, you might notice that the iterators in this assignment extend the `Operator` class instead of implementing the DbIterator interface. Because the implementation of next/hasNext is often repetitive, annoying, and error-prone, `Operator` implements this logic generically, and only requires that you implement a simpler readNext. Feel free to use this style of implementation, or just implement the `DbIterator` interface if you prefer. To implement the DbIterator interface, remove `extends Operator` from iterator classes, and in its place put `implements DbIterator`.

2. SimpleDB Architecture and Implementation Guide

2.1. *Filter and Join*

Recall that SimpleDB DbIterator classes implement the operations of the relational algebra. You will now implement two operators that will enable you to perform queries that are slightly more interesting than a table scan.

- *Filter*: This operator only returns tuples that satisfy a `Predicate` that is specified as part of its constructor. Hence, it filters out any tuples that do not match the predicate.
- *Join*: This operator joins tuples from its two children according to a `JoinPredicate` that is passed in as part of its constructor. We require a simple nested loops join implementation and a hash join implementation respectively, but you may explore more interesting join implementations. Describe your implementation in your writeup.

Exercise 1. Implement the skeleton methods in:

- src/simpledb/Predicate.java
- src/simpledb/JoinPredicate.java
- src/simpledb/Filter.java
- src/simpledb/Join.java
- src/simpledb/HashEquiJoin.java

At this point, your code should pass the unit tests in PredicateTest, JoinPredicateTest, FilterTest, JoinTest, and HashEquiJoinTest. Furthermore, you should be able to pass the system tests FilterTest, JoinTest and HashEquiJoinTest.

2.2. *Aggregates*

An additional SimpleDB operator implements basic SQL aggregates with a `GROUP BY` clause. You should implement the five SQL aggregates (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) and support grouping. You only need to support aggregates over a single field, and grouping by a single field.

In order to calculate aggregates, we use an `Aggregator` interface which merges a new tuple into the existing calculation of an aggregate. The `Aggregator` is told during construction what operation it should use for aggregation. Subsequently, the client code should call `Aggregator.mergeTupleIntoGroup()` for every tuple in the child iterator. After all tuples have been merged, the client can retrieve 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)`.

Note that this 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.

Exercise 2. Implement the skeleton methods in:

- src/simpledb/IntegerAggregator.java
- src/simpledb/StringAggregator.java
- src/simpledb/Aggregate.java

At this point, your code should pass the unit tests IntegerAggregatorTest, StringAggregatorTest, and AggregateTest. Furthermore, you should be able to pass the AggregateTest system test.

2.3. *HeapFile Mutability*

Now, we will begin to implement methods to support modifying tables. We begin at the level of individual pages and files. There are two main sets of operations: adding tuples and removing tuples.

Removing tuples: To remove a tuple, you will need to implement `deleteTuple`. Tuples contain `RecordIDs` which allow you to find the page they reside on, so this should be as simple as locating the page a tuple belongs to and modifying the headers of the page appropriately.

Adding tuples: The `insertTuple` method in `HeapFile.java` is responsible for adding a tuple to a heap file. To add a new tuple to a HeapFile, you will have to find a page with an empty slot. If no such pages exist in the HeapFile, you need to create a new page and append it to the physical file on disk. You will need to ensure that the RecordID in the tuple is updated correctly.

Exercise 3. Implement the remaining skeleton methods in:

- src/simpledb/HeapPage.java
- src/simpledb/HeapFile.java
- (Note that you do not necessarily need to implement writePage at this point).

To implement HeapPage, you will need to modify the header bitmap for methods such as insertTuple() and deleteTuple(). You may find that the getNumEmptySlots() and isSlotUsed() methods we asked you to implement in PA 1 serve as useful abstractions. Note that there is a markSlotUsed method provided as an abstraction to modify the filled or cleared status of a tuple in the page header.

filled or cleared status of a tuple in the page header.

Note that it is important that the HeapFile.insertTuple() and HeapFile.deleteTuple() methods access pages using the BufferPool.getPage() method; otherwise, your implementation of transactions in the next assignment will not work properly.

Implement the following skeleton methods in src/simpledb/BufferPool.java:

- insertTuple()
- deleteTuple()

These methods should call the appropriate methods in the HeapFile that belong to the table being modified (this extra level of indirection is needed to support other types of files — like indices — in the future).

At this point, your code should pass the unit tests in HeapPageWriteTest and HeapFileWriteTest..

2.4. *Insertion and deletion*

Now that you have written all of the HeapFile machinery to add and remove tuples, you will implement the `Insert` and `Delete` operators.

For plans that implement `insert` and `delete` queries, the top-most operator is a special `Insert` or `Delete` operator that modifies the pages on disk. These operators return the number of affected tuples. This is implemented by returning a single tuple with one integer field, containing the count.

- *Insert*: This operator adds the tuples it reads from its child operator to the `tableid` specified in its constructor. It should use the `BufferPool.insertTuple()` method to do this.
- *Delete*: This operator deletes the tuples it reads from its child operator from the `tableid` specified in its constructor. It should use the `BufferPool.deleteTuple()` method to do this.

Exercise 4. Implement the skeleton methods in:

- src/simpledb/Insert.java
- src/simpledb/Delete.java

At this point, your code should pass the unit tests in InsertTest. We have not provided unit tests for `Delete`. Furthermore, you should be able to pass the InsertTest and DeleteTest system tests.

2.5. *Query walkthrough*

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

For more extensive examples of query operations, you may find it helpful to browse the unit tests for joins, filters, and aggregates.

```
package simpledb;

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();

        }

    }
```

```
}
```

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).

Once we have finished initializing the database system, 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 e.g., 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.

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made with a little more details. If you used something other than a nested-loops join, describe the tradeoffs of the algorithm you chose.
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the assignment, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

Please indicate clearly who you worked with (if anyone) and the division of the work on your writeup.

3.2. Submitting your assignment

Please also submit your writeup as a PDF or plain text file (.txt). Please do not submit a .doc or .docx. Use the same submission procedure as before.

Make sure your code is packaged so the instructions outlined in section 3.4 work.

3.3. Submitting a bug

Si eDB is a relatively complex piece of code. It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.

We ask you, therefore, to do this assignment with an adventurous mindset. Don't get mad if something is not clear, or even wrong; rather, try to figure it out yourself or send us a friendly email. Please submit bug reports to either the TF or the instructor. When you do, please try to include:

- A description of the bug.
- A .java file we can drop in the `test/simpledb` directory, compile, and run.
- A .txt file with the data that reproduces the bug. We should be able to convert it to a .dat file using `HeapFileEncoder`.

You can also post on piazza if you feel you have run into a bug.

3.4 Grading

85% of your grade will be based on whether or not your code passes the system test suite we will run over it. These tests will be a superset of the tests we have provided. Before handing in your code, you should make sure it produces no errors (passes all of the tests) from both ant test and ant systemtest.

Important: before testing, we will replace your build.xml, HeapFileEncoder.java, and the entire contents of the test/ directory with our version of these files! This means you cannot change the format of .dat files! You should therefore be careful changing our APIs. This also means you need to test whether your code compiles with our test programs. In other words, we will untar your tarball, replace the files mentioned above, compile it, and then grade it. It will look roughly like this:

```
[replace build.xml, HeapFileEncoder.java, and test]

$ ant test

$ ant systemtest

[additional tests]
```

If any of these commands fail, we'll be unhappy, and, therefore, so will your grade.

An additional **15%** of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

We've had a lot of fun designing this assignment, and we hope you enjoy hacking on it!