CSI 508. Database Systems I – Fall 2016

Programming Assignment I

The total grade for this assignment is 100 points. The deadline for this assignment is **11:59 PM, October 10, 2016**. *Submissions after this deadline will not be accepted.* Students are required to enter the UAlbany Blackboard system and then upload a .zip file (in the form of [last name]_[first name].zip) that contains the Eclipse project directory and a short document that describes:

- the methods implemented
- any missing or incomplete elements of the code
- any changes made to the original API
- the total amount of time spent for this assignment (also the days when you worked on this assignment and the amount of time spent for each day)
- suggestions or comments if any

In this programming assignment, you need to extend a Java-based database management system called SimpleDB by implementing the core modules for accessing data stored on disk. We have provided you with a set of incomplete classes and interfaces (see those in the source code with methods that currently throw `UnsupportedOperationException`s). You will need to write code for these classes. Your code will be graded by running a set of system tests written using JUnit and then examining your code. Details of JUnit can be found at `http://junit.org`. Passing JUnit tests does not necessarily guarantee that your implementation is correct and efficient. Please strive to write correct and efficient code. If you have questions, please contact the TA or the instructor.

You first need to install/run Eclipse on your machine and import the "`simpledb-1`" project (see Appendix A). This assignment provides an Eclipse project that contains source code and test suites. Refer to Appendix B for the details of running the test suites. You can also generate HTML-based API documents from the source code using `javadoc`. Refer to Appendix C for the details of using `javadoc`. The architecture of SimpleDB is described in Appendix D. The remainder of this document describes the components that you need to implement.

Part 1. Fields and Tuples (20 points)

Each `Tuple` in SimpleDB consists of a collection of `Field` objects. `Field` is an interface that different data types (e.g., integer, string) implement. `Tuple`s also have a type (or schema), represented by a `TupleDesc` (tuple descriptor) object. This object consists of a collection of `Type` objects, one per field in the tuple, each of which describes the type of the corresponding field.

In this part, you need to complete *two methods* of the `TupleDesc` class and *one method* of the `Tuple` class that throw `UnsupportedOperationException`s. Once you have completed these classes, your code will pass the unit tests `TupleDescTest` and `TupleTest`.

Part 2. Catalog (20 points)

The catalog in SimpleDB maintains the tables and schemas of the tables that are currently in the database. You need to implement, in `Catalog.java`, *three methods* (`getTableId(String name)`, `getTupleDesc(int tableid)` and `getDbFile(int tableid)`). These methods need to use

member variables (`name2tableID`, `tableID2desc`, `tableID2dbFile`) of the `Catalog` class. Associated with each table is a `TupleDesc` object that allows operators to determine the types and number of fields in a table.

The global catalog is a single instance of `Catalog` that is allocated for the entire SimpleDB system. This global catalog can be accessed via the `Database` class's `getCatalog()` method.

Once you have completed this part, your code will pass the unit tests in `CatalogTest`.

Part 3. BufferPool (20 points)

The buffer pool is responsible for caching pages in memory that have been recently read from disk. All operators read and write pages from various files on disk through this buffer pool. The buffer pool consists of a fixed number of pages, defined by the `numPages` parameter to the `BufferPool` constructor. For this assignment, you need to implement the `getPage(TransactionId tid, PageId pid, Permissions perm)` method of `BufferPool` (eventually used by the `SeqScan` operator which sequentially reads pages from a `DbFile`). The `BufferPool` needs to keep/maintain up to `numPages` pages using the `pages` member variable. The type of this `pages` member variable is `Hashtable<PageId, Page>`. In this assignment, if more than `numPages` requests are made for different pages, then a `DbException` must be thrown (instead of implementing an eviction policy). If the `pid` parameter of the `getPage(TransactionId tid, PageId pid, Permissions perm)` method is already registered in the `pages` hash table, you need to return the `Page` instance registered with the `pid` in the `pages` hash table. Otherwise (i.e., if the `pid` parameter is not yet registered in the `pages` hash table), you need to read the wanted page from an appropriate `DbFile`. For this, obtain the `DbFile` by using `Database.getCatalog().getDbFile(pid.getTableId())` and then call the `readPage(PageId pid)` method on the `DbFile`.

The `Database` class provides a static method, `getBufferPool()`, which returns a reference to the single `BufferPool` instance for the entire SimpleDB system. There is no unit test for `BufferPool`. The functionality you implemented will be tested after the `HeapFile` class is completed as explained below.

Part 4. HeapPage access method (20 points)

Access methods provide a way to read or write data on disk that is arranged in a specific way. Common access methods include heap files (unsorted files of tuples) and B+-trees. For this assignment, you need to implement only the heap file access method, and we have written some of the code for you.

In SimpleDB, there is one `HeapFile` object for each table in the database. A `HeapFile` object is arranged into a set of pages. Pages of a `HeapFile` are of the `HeapPage` type which implements the `Page` interface. A `HeapPage` instance has a byte array for storing tuples and a header (see the `data` byte array in `HeapPage.java`). The size of the byte array is defined by the constant `BufferPool.PAGE_SIZE`. The header of a page consists of (1) a 4-byte integer representing the number of entries in the page for storing tuples, and (2) a series of 4-byte integers (one for each tuple) each of which represents where the corresponding tuple is stored in the page (i.e., the location of the corresponding tuple within the page). Between the header and the tuples, each page has some free space where more tuples and header entries can be added. For further details, refer to Section 10.5.2 of the textbook. If a tuple is deleted, then the corresponding entry (for storing the location of the tuple) in the header is set to -1 to indicate that there is no associated tuple. Pages are kept in the buffer pool after being read from disk via the `HeapFile` class's `readPage(PageId pid)` method.

In this part, you need to implement the `getTuple(int entryID)`, `entryCount()`, `tupleLocation(int entryID)`, and `iterator()` methods of the `HeapPage` class. The `getTuple(int entryID)` method must read the tuple specified by `entryID` from the `data` byte array and then return that tuple. For example, when the byte array `data` stores a header and tuples as described in the previous

paragraph, `getTuple(0)` must return the tuple at entry `0` and `getTuple(1)` must return the tuple at entry `1`. To read the tuple specified by `entryID`, the `getTuple(int entryID)` method must obtain the location of the wanted tuple in the `data` array by calling `tupleLocation(entryID)`. Then, the tuple specified by `entryID` can be obtained by calling `createTuple(DataInputStream in)` where `in` is a `DataInputStream` that begins at the location of the tuple in the byte array `data` (i.e., `in = new DataInputStream(new ByteArrayInputStream(data, location, data.length - location))` where `location` is the location of the tuple specified by `entryID`). After obtaining the tuple, call `setRecordId(new RecordId(pid, entryID))` on the tuple to register the ID of the page and the entry storing the tuple.

The `entryCount()` method must return the number of entries in the current `HeapPage`. This method can be simply implemented by making it return the value of `readInt(data, 0)` because `readInt(byte[] data, int location)` method (already provided in `HeapPage.java`) reads a 4-byte integer at the specified location (`location`) of the given byte array (`data`) and the integer at location `0` in `data` stores the number of entries in the `HeapPage`.

The `tupleLocation(int entryID)` can also be implemented by making it return `readInt(data, 4 + 4 * entryID)` because the first 4-byte integer in `data` represents the number of entries and the subsequent 4-byte integers represent the location of the tuple at entry `0`, the location of the tuple at entry `1`, and so on. For example, `tupleLocation(1)` must return the integer at location 8 in the `data` array.

The `iterator()` method must be able to iterate over the tuples (excluding those deleted) in the page on which the `iterator()` method is invoked. To find the number of entries in the current `HeapPage`, use `entryCount()`. To get the tuple at an entry specified by `entryID`, use `getTuple(int entryID)`. `getTuple(int entryID)` returns `null` if the specified entry does not contain any tuple due to the deletion of the previous tuple. Feel free to add an auxiliary class or data structure for this `iterator()` method.

At this point, your code needs to pass the unit tests in `HeapPageReadTest`.

Part 5. HeapFile access method (20 points)

After you have implemented `HeapPage`, you need to write two methods for the `HeapFile` class to read a page from a file stored on disk. In particular, you need to implement the `readPage(PageId pid)` method to read a page from disk. This method must use *random access* to the file in order to directly read and write the page specified by `pid` (do not sequentially access pages from the beginning of that file). For this random access, the method must calculate the correct location (offset) of the wanted page in the file (note that the size of each page is `BufferPool.PAGE_SIZE`). You should not call `BufferPool`'s methods when reading a page from disk.

You also need to implement `HeapFile`'s `iterator(TransactionId tid)` method, which should iterate through the tuples of each page in the `HeapFile`. The iterator must use the `BufferPool` class's `getPage(TransactionId tid, PageId pid, Permissions perm)` method to access pages in the `HeapFile`. This method must work properly even if the file is too large to fit in main memory.

At this point, your code needs to pass the unit tests in `HeapFileReadTest` and `ScanTest`.

Final Remark

Operators are responsible for the actual execution of the query plan. They implement the operations of the relational algebra. In SimpleDB, operators are iterator based; each operator implements the `DbIterator` interface. Operators are connected together into a plan by passing lower-level operators into the constructors of higher-level operators, i.e., by "chaining them together". Special access method operators at the leaves of the plan are responsible for reading data from the disk (and hence do not have any operators below them).

At the top of the plan, the program interacting with SimpleDB simply calls the `getNext()` method on the root operator. This operator then calls `getNext()` on its children, and so on, until

these leaf operators are called. They fetch tuples from disk and pass them up the tree (as return arguments to `getNext()`). Tuples propagate up the plan in this way until they are output at the root or combined or rejected by another operator in the plan.

In this assignment, we use `src/simpledb/SeqScan.java`, which is already implemented. This operator sequentially scans all of the tuples from the pages of the table specified by the `tableid` in the constructor. This operator accesses tuples through the `DbFile` class's `iterator(TransactionId tid)` method.

Now, let's see how the above components are connected together to process a simple query. The following code implements a simple selection query over a data file consisting of three columns of integers. In the code, the file `some_data_file.dat` is a binary representation of the pages from this file. This code is equivalent to the SQL statement `SELECT * FROM some_data_file`.

```
// 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 descriptor = new TupleDesc(types, names);

// create the table, associate it with some_data_file.dat
// and tell the catalog about the schema of this table.
HeapFile table1 = new HeapFile(new File("some_data_file.dat"));
Database.getCatalog().addTable(table1, descriptor);

// construct the query: we use a simple SeqScan, which spoonfeeds
// tuples via its iterator.
TransactionId tid = new TransactionId();
SeqScan f = new SeqScan(tid, table1.id());

// and run it
f.open();
while (f.hasNext()) {
    Tuple tup = f.next();
    System.out.println(tup);
}
f.close();

Database.getBufferPool().transactionComplete(tid);
```

The above code creates a table that has three integer fields. To express this, we create a `TupleDesc` object and pass it an array of `Type` objects, and optionally an array of `String` field names. Once we have created this `TupleDesc`, we initialize a `HeapFile` object representing the table stored in `some_data_file.dat`. Once we have created the table, we add it to the catalog.

After initializing the database system, we create a query plan. Our plan consists only of the `SeqScan` operator that scans the tuples from disk. In general, operators are instantiated with references to the appropriate table (in the case of `SeqScan`) or child operator (e.g., in the case of `Filter`). The test program then repeatedly calls `hasNext()` and `next()` on the `SeqScan` operator. As tuples are output from the `SeqScan` operator, they are printed out on the command line.

This is all! We hope this assignment will help you understand how real database management systems work.

Appendix A. Installing Eclipse and Importing a Java Project

1. Visit:

   `http://www.eclipse.org/downloads/`

2. From the web site, download the eclipse installer (those for Linux, Windows, and Mac OS X are available) and then choose "Eclipse IDE for Java Developers" and install it.

3. After finishing installation, start Eclipse.

4. When Eclipse runs for the first time, it asks the user to choose the workspace location. You may use the default location.

5. 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 "2016_p1.zip" file contained in this assignment package.

Appendix B. Using Ant in Eclipse

For this assignment, you need to use the `Ant` build tool to compile the code and run tests. Ant is similar to `make`, but the build file is written in XML and is somewhat better suited to Java code. Further details about `Ant` is available at `http://ant.apache.org/`.

To help you during development, a set of unit tests are provided in addition to the end-to-end tests for grading. The following table shows the ant commands that you can use for this assignment:

Command	Description
`ant`	Build the default target (for simpledb, this is dist).
`ant -projecthelp`	List all the targets in `build.xml` with descriptions.
`ant dist`	Compile the code in `src` and package it in `dist/simpledb.jar`.
`ant test`	Compile and run all the unit tests.
`ant runtest -Dtest=testname`	Run the unit test named `testname`.
`ant systemtest`	Compile and run all the system tests.
`ant runsystest -Dtest=testname`	Compile and run the system test named `testname`.

Running Ant Build Targets in Eclipse

If you want to run commands such as "`ant test`" or "`ant systemtest`," right click on `build.xml` in the Package Explorer. Select "Run As", and then "Ant Build..." (note: don't be confused with "Ant Build", which would not show a set of build targets to run). Then, in the "Targets" tab of the next screen, check off the targets you want to run (probably "`dist`" and one of "`test`" or "`systemtest`"). This should run the build targets and show you the results in Eclipse's console window.

If you run the unit tests using the `test` build target, you will see output similar to:

```
Buildfile: /Users/jhh/Documents/workspace/simpledb-1/build.xml

compile:

testcompile:

systemtest:
    [junit] Running simpledb.systemtest.ScanTest
    [junit] Testsuite: simpledb.systemtest.ScanTest
    [junit] Tests run: 3, Failures: 0, Errors: 3, Skipped: 0, Time elapsed: 0.13 sec
    [junit] Tests run: 3, Failures: 0, Errors: 3, Skipped: 0, Time elapsed: 0.13 sec
    [junit] Testcase: testSmall took 0.03 sec
    [junit]  Caused an ERROR
    [junit] Implement this
    [junit] java.lang.UnsupportedOperationException: Implement this
    [junit]  at simpledb.Catalog.getDbFile(Catalog.java:105)
    [junit]  at simpledb.SeqScan.<init>(SeqScan.java:28)
    [junit]  at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:106)
    [junit]  at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:100)
    [junit]  at simpledb.systemtest.ScanTest.validateScan(ScanTest.java:32)
    [junit]  at simpledb.systemtest.ScanTest.testSmall(ScanTest.java:43)
    [junit] Testcase: testRewind took 0.014 sec
```

```
    [junit]  Caused an ERROR
    [junit] Implement this
    [junit] java.lang.UnsupportedOperationException: Implement this
    [junit]  at simpledb.Catalog.getDbFile(Catalog.java:105)
    [junit]  at simpledb.SeqScan.<init>(SeqScan.java:28)
    [junit]  at simpledb.systemtest.ScanTest.testRewind(ScanTest.java:52)
    [junit] Testcase: testCache took 0.067 sec
    [junit]  Caused an ERROR
    [junit] Implement this
    [junit] java.lang.UnsupportedOperationException: Implement this
    [junit]  at simpledb.Catalog.getDbFile(Catalog.java:105)
    [junit]  at simpledb.SeqScan.<init>(SeqScan.java:28)
    [junit]  at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:106)
    [junit]  at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:100)
    [junit]  at simpledb.systemtest.ScanTest.testCache(ScanTest.java:98)

BUILD FAILED
/Users/jhh/Documents/workspace/simpledb-1/build.xml:117: The following error occurred while executing this
/Users/jhh/Documents/workspace/simpledb-1/build.xml:56: Test simpledb.systemtest.ScanTest failed

Total time: 899 milliseconds
```

The above output indicates that three errors occurred. This is because the current code doesn't yet work. As you complete more components of the system, you will work towards passing additional unit tests.

Running Individual Unit and System Tests in Eclipse

To run a unit test or system test (both are JUnit tests and can be initialized the same way), go to the Package Explorer tab on the left side of your screen. Under the "`simpledb-1`" project, open the "`test`" directory. Unit tests are found in the "`simpledb`" package, and system tests are found in the "`simpledb.systemtests`" package. To run one of these tests, select the test (they are all called `*Test.java` - don't select `TestUtil.java` or `SystemTestUtil.java`), right click on it, select "Run As", and select "JUnit Test". This will bring up a JUnit tab, which will tell you the status of the individual tests within the JUnit test suite, and will show you exceptions and other errors that will help you debug problems.

Appendix C. Creating API documents using `javadoc`

One nice feature of Java is its support for "documentation comments", or "`javadoc`" comments, which you can use to automatically produce documentation for your code. Javadoc comments start with "`/**`". Inside a javadoc comment, there are some special symbols, like `@param` and `@return` as in `simpledb-1/src/simpledb/Tuple.java`.

You can create HTML-based API documents from the source as follows:

1. Click the "`simpledb-1`" icon in the Navigator or Project Explorer window.
2. Select "Generate Javadoc" from the "Project" menu.
3. In the "Generate Javadoc" dialog box, press the "Finish" button.

As it runs, it tells you that it's generating various things. When it's finished, a few new folders should appear in your project: `doc`, `doc.javadoc`, and doc.resources. Poke around and see what got generated (to open the newly created HTML documentation files in a web browser window, just double-click them; you can start with "`index.html`"). It's a lot of stuff for such a small effort in documentation!

Appendix D. SimpleDB Architecture

SimpleDB consists of:

- Classes that represent fields, tuples, and tuple schemas;
- Classes that apply predicates and conditions to tuples;
- One or more access methods (e.g., heap files) that store relations on disk and provide a way to iterate through tuples of those relations;
- A collection of operator classes (e.g., select, join, insert, delete, etc.) that process tuples;
- A buffer pool that caches active tuples and pages in memory and handles concurrency control and transactions (neither of which you need to worry about for this assignment);
- A catalog that stores information about available tables and their schemas; and
- The `Database` class, which provides access to a collection of static objects that are the global state of the database. In particular, this includes methods to access the catalog, the buffer pool, and the log file.