6.830/6.814 Lab 1: SimpleDB

Assigned: Wed, Feb 24

Due: Wed, Mar 10 11:59 PM EDT

In the lab assignments in 6.830 you will write a basic database management system called SimpleDB. For this lab, you will focus on implementing the core modules required to access stored data on disk; in future labs, you will add support for various query processing operators, as well as transactions, locking, and concurrent queries.

SimpleDB is written in Java. We have provided you with a set of mostly unimplemented classes and interfaces. You will need to write the code for these classes. We will grade your code by running a set of system tests written using JUnit. We have also provided a number of unit tests, which we will not use for grading but that you may find useful in verifying that your code works. We also encourage you to develop your own test suite in addition to our tests.

The remainder of this document describes the basic architecture of SimpleDB, gives some suggestions about how to start coding, and discusses how to hand in your lab.

We **strongly recommend** that you start as early as possible on this lab. It requires you to write a fair amount of code!

<!--

0. Find bugs, be patient, earn candy bars

SimpleDB 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 It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.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 lab 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. We promise to help out by posting bug fixes, new commits to the HW repo, etc., as bugs and issues are reported.

...and if you find a bug in our code, we`ll give you a candy bar (see [Section 3.3](#bugs))!

-->

0. Environment Setup

Start by downloading the code for lab 1 from the course GitHub repository by following the instructions here.

These instructions are written for Athena or any other Unix-based platform (e.g., Linux, MacOS, etc.) Because the code is written in Java, it should work under Windows as well, although the directions in this document may not apply.

We have included Section 1.2 on using the project with Eclipse or IntelliJ.

1. Getting started

SimpleDB uses 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. Most modern Linux distributions include Ant. Under Athena, it is included in the `sipb` locker, which you can get to by typing `add sipb` at the Athena prompt. Note that on some versions of Athena you must also run `add -f java` to set the environment correctly for Java programs. See the Athena documentation on using Java for more details.

To help you during development, we have provided a set of unit tests in addition to the end-to-end tests that we use for grading. These are by no means comprehensive, and you should not rely on them exclusively to verify the correctness of your project (put those 6.170 skills to use!).

To run the unit tests use the `test` build target:

```
$ cd [project-directory]
$ # run all unit tests
$ ant test
$ # run a specific unit test
$ ant runtest -Dtest=TupleTest
```

You should see output similar to:

```
 build output...

test:
    [junit] Running simpledb.CatalogTest
    [junit] Testsuite: simpledb.CatalogTest
    [junit] Tests run: 2, Failures: 0, Errors: 2, Time elapsed:
0.037 sec
    [junit] Tests run: 2, Failures: 0, Errors: 2, Time elapsed:
0.037 sec

 ... stack traces and error reports ...
```

The output above indicates that two errors occurred during compilation; this is because the code we have given you
doesn't yet work. As you complete parts of the lab, you will work towards passing additional unit tests.

If you wish to write new unit tests as you code, they should be added to the `test/simpledb` directory.

For more details about how to use Ant, see the [manual](http://ant.apache.org/manual/). The [Running Ant](http://ant.apache.org/manual/running.html) section provides details about using the `ant` command. However, the quick reference table below should be sufficient for working on the labs.

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`.

If you are under windows system and don't want to run ant tests from command line, you can also run them from eclipse.
Right click build.xml, in the targets tab, you can see "runtest" "runsystest" etc. For example, select runtest would be
equivalent to "ant runtest" from command line. Arguments such as "-Dtest=testname" can be specified in the "Main" Tab, "
Arguments" textbox. Note that you can also create a shortcut to runtest by copying from build.xml, modifying targets and
arguments and renaming it to, say, runtest_build.xml.

1.1. Running end-to-end tests

We have also provided a set of end-to-end tests that will eventually be used for grading. These tests are structured as
JUnit tests that live in the `test/simpledb/systemtest` directory. To run all the system tests, use
the `systemtest` build target:

```
$ ant systemtest

 ... build output ...

    [junit] Testcase: testSmall took 0.017 sec
    [junit]     Caused an ERROR
    [junit] expected to find the following tuples:
    [junit]     19128
    [junit]
```

```
    [junit] java.lang.AssertionError: expected to find the
following tuples:
    [junit]     19128
    [junit]
    [junit]     at
simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:
122)
    [junit]     at
simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:
83)
    [junit]     at
simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:
75)
    [junit]     at
simpledb.systemtest.ScanTest.validateScan(ScanTest.java:30)
    [junit]     at
simpledb.systemtest.ScanTest.testSmall(ScanTest.java:40)

 ... more error messages ...
```

This indicates that this test failed, showing the stack trace where the error was detected. To debug, start by reading the source code where the error occurred. When the tests pass, you will see something like the following:

```
$ ant systemtest

 ... build output ...

    [junit] Testsuite: simpledb.systemtest.ScanTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed:
7.278 sec
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed:
7.278 sec
    [junit]
    [junit] Testcase: testSmall took 0.937 sec
    [junit] Testcase: testLarge took 5.276 sec
    [junit] Testcase: testRandom took 1.049 sec

BUILD SUCCESSFUL
Total time: 52 seconds
```

1.1.1 Creating dummy tables

It is likely you'll want to create your own tests and your own data tables to test your own implementation of SimpleDB.
You can create any `.txt` file and convert it to a `.dat` file in SimpleDB's `HeapFile` format using the
command:

```
$ java -jar dist/simpledb.jar convert file.txt N
```

where `file.txt` is the name of the file and `N` is the number of columns in the file. Notice that
`file.txt` has to be in the following format:

```
int1,int2,...,intN
int1,int2,...,intN
int1,int2,...,intN
int1,int2,...,intN
```

...where each intN is a non-negative integer.

To view the contents of a table, use the `print` command:

```
$ java -jar dist/simpledb.jar print file.dat N
```

where `file.dat` is the name of a table created with the `convert` command, and `N` is the number
of columns in the file.

1.2. Working with an IDE

IDEs (Integrated Development Environments) are graphical software development environments that may help you manage
larger projects. We provide instructions for setting up both
Eclipse and IntelliJ. The instructions we provide for
Eclipse were generated by using Eclipse for Java Developers (not the enterprise edition) with Java 1.7. For IntelliJ, we
are using the Ultimate edition, which you can get with an education license through your mit.edu
account here. We strongly encourage you to set up and learn
one of the IDEs for this project.

Preparing the Codebase

Run the following command to generate the project file for IDEs:

```
ant eclipse
```

Setting the Lab Up in Eclipse

- Once Eclipse is installed, start it, and note that the first screen asks you to select a location for your workspace (
  we will refer to this directory as $W). Select the directory containing your simple-db-hw repository.
- In Eclipse, select File->New->Project->Java->Java Project, and push Next.
- Enter "simple-db-hw" as the project name.
- On the same screen that you entered the project name, select "Create project from existing source," and browse to
  $W/simple-db-hw.
- Click finish, and you should be able to see "simple-db-hw" as a new project in the Project Explorer tab on the

left-hand side of your screen. Opening this project reveals the directory structure discussed above - implementation
code can be found in "src," and unit tests and system tests found in "test."

Note: that this class assumes that you are using the official Oracle release of Java. This is the default on MacOS
X, and for most Windows Eclipse installs; but many Linux distributions default to alternate Java runtimes (like OpenJDK)
. Please download the latest Java8 updates
from Oracle Website, and use that Java version. If
you don't switch, you may see spurious test failures in some of the performance tests in later labs.

Running Individual Unit and System Tests

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 "simple-db-hw" 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.

Running Ant Build Targets

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: select the option with the ellipsis (...), otherwise you won't be
presented with 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.

Setting the Lab Up in IntelliJ

IntelliJ is a more modern Java IDE that is popular and more intuitive to use by some accounts. To use IntelliJ, first
install it and open the application. Similar to Eclipse, under Projects, select Open and navigate to your project root.
Double-click on the .project file (you may need to configure your operating system to reveal hidden files to see it),
and click "open as project". IntelliJ has tool window support with Ant that you may want to setup according to
instructions here, but this is not essential to development.
You can find a detailed walkthrough of IntelliJ features here

1.3. Implementation hints

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

You will need to fill in any piece of code that is not implemented. It will be obvious where we think you should write
code. You may need to add private methods and/or helper classes. You may change APIs, but make sure
our grading tests still run and make sure to mention, explain, and defend your decisions in your writeup.

In addition to the methods that you need to fill out for this lab, the class interfaces contain numerous methods that
you need not implement until subsequent labs. These will either be indicated per class:

```
// Not necessary for lab1.
public class Insert implements DbIterator {
```

or per method:

```
public boolean deleteTuple(Tuple t)throws DbException{
        // some code goes here
        // not necessary for lab1
        return false;
        }
```

The code that you submit should compile without having to modify these methods.

We suggest exercises along this document to guide your implementation, but you may find that a different order makes
more sense for you.

Here's a rough outline of one way you might proceed with your SimpleDB implementation:

- Implement the classes to manage tuples, namely Tuple, TupleDesc. We have already implemented Field, IntField,StringField, and Type for you. Since you only need to support integer and (fixed length) string fields and fixedlength tuples, these are straightforward.
- Implement the Catalog (this should be very simple).
- Implement the BufferPool constructor and the getPage() method.
- Implement the access methods, HeapPage and HeapFile and associated ID classes. A good portion of these files has already been written for you.
- Implement the operator SeqScan.
- At this point, you should be able to pass the ScanTest system test, which is the goal for this lab.

Section 2 below walks you through these implementation steps and the unit tests corresponding to each one in more
detail.

1.4. Transactions, locking, and recovery

As you look through the interfaces we have provided you, you will see a number of references to locking, transactions,
and recovery. You do not need to support these features in this lab, but you should keep these parameters in the
interfaces of your code because you will be implementing transactions and locking in a future lab. The test code we have
provided you with generates a fake transaction ID that is passed into the operators of the query it runs; you should
pass this transaction ID into other operators and the buffer pool.

2. SimpleDB Architecture and Implementation Guide

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 lab); and,
- A catalog that stores information about available tables and their schemas.

SimpleDB does not include many things that you may think of as being a part of a "database." In particular, SimpleDB
does not have:

- (In this lab), a SQL front end or parser that allows you to type queries directly into SimpleDB. Instead, queries are built up by chaining a set of operators together into a hand-built query plan (see Section 2.7).
  We will provide a simple parser for use in later labs.
- Views.
- Data types except integers and fixed length strings.
- (In this lab) Query optimizer.
- (In this lab) Indices.

In the rest of this Section, we describe each of the main components of SimpleDB that you will need to implement in this
lab. You should use the exercises in this discussion to guide your implementation. This document is by no means a
complete specification for SimpleDB; you will need to make decisions about how to design and implement various parts of
the system. Note that for Lab 1 you do not need to implement any operators (e.g., select, join, project) except
sequential scan. You will add support for additional operators in future labs.

2.1. The Database Class

The Database class 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 list of all the tables in the database), the buffer pool (
the collection of database file pages that are currently resident in memory), and the log file. You will not need to
worry about the log file in this lab. We have implemented the Database class for you. You should take a look at this
file as you will need to access these objects.

2.2. Fields and Tuples

Tuples in SimpleDB are quite basic. They consist of a collection of `Field` objects, one per field in the `Tuple`. `Field` is an interface that different data types (e.g., integer, string) implement. `Tuple` objects are created by the underlying access methods (e.g., heap files, or B-trees), as described in the next section. Tuples also have a type (or schema), called a _tuple descriptor_, represented by a `TupleDesc` 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.

Exercise 1

Implement the skeleton methods in:

- src/java/simpledb/storage/TupleDesc.java
- src/java/simpledb/storage/Tuple.java

At this point, your code should pass the unit tests TupleTest and TupleDescTest. At this point, modifyRecordId() should
fail because you havn't implemented it yet.

2.3. Catalog

The catalog (class `Catalog` in SimpleDB) consists of a list of the tables and schemas of the tables that are currently
in the database. You will need to support the ability to add a new table, as well as getting information about a
particular table. 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 process. The global
catalog can be retrieved via the method `Database.getCatalog()`, and the same goes for the global buffer pool (
using `Database.getBufferPool()`).

Exercise 2

Implement the skeleton methods in:

- src/java/simpledb/common/Catalog.java

At this point, your code should pass the unit tests in CatalogTest.

2.4. BufferPool

The buffer pool (class `BufferPool` in SimpleDB) 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 the buffer pool. It consists of a fixed number of pages, defined by the `numPages` parameter to the `BufferPool` constructor. In later labs, you will implement an eviction policy. For this lab, you only need to implement the constructor and the `BufferPool.getPage()` method used by the SeqScan operator. The BufferPool should store up to `numPages` pages. For this lab, if more than `numPages` requests are made for different pages, then instead of implementing an eviction policy, you may throw a DbException. In future labs you will be required to implement an eviction policy.

The `Database` class provides a static method, `Database.getBufferPool()`, that returns a reference to the single
BufferPool instance for the entire SimpleDB process.

Exercise 3

Implement the `getPage()` method in:

- src/java/simpledb/storage/BufferPool.java

We have not provided unit tests for BufferPool. The functionality you implemented will be tested in the implementation
of HeapFile below. You should use the `DbFile.readPage` method to access pages of a DbFile.

<!--

Notice that `BufferPool` asks you to implement
a `flush_all_pages()` method. This is not something you would ever
need in a real implementation of a buffer pool. However, we need this method
for testing purposes. You really should never call this method from anywhere
in your code.
-->

2.5. HeapFile access method

Access methods provide a way to read or write data from 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 will only implement a heap file
access method, and we have written some of the code for you.

A `HeapFile` object is arranged into a set of pages, each of which consists of a fixed number of bytes for storing
tuples, (defined by the constant `BufferPool.DEFAULT_PAGE_SIZE`), including a header. In SimpleDB, there is
one `HeapFile` object for each table in the database. Each page in a `HeapFile` is arranged as a set of slots, each of
which can hold one tuple (tuples for a given table in SimpleDB are all of the same size). In addition to these slots,
each page has a header that consists of a bitmap with one bit per tuple slot. If the bit corresponding to a particular
tuple is 1, it indicates that the tuple is valid; if it is 0, the tuple is invalid (e.g., has been deleted or was never
initialized.) Pages of `HeapFile` objects are of type `HeapPage` which implements the `Page` interface. Pages are
stored in the buffer pool but are read and written by the `HeapFile` class.

SimpleDB stores heap files on disk in more or less the same format they are stored in memory. Each file consists of page
data arranged consecutively on disk. Each page consists of one or more bytes representing the header, followed by the _
page size_ bytes of actual page content. Each tuple requires *tuple size* * 8 bits for its content and 1 bit for the
header. Thus, the number of tuples that can fit in a single page is:

`_tuples per page_ = floor((_page size_ * 8) / (_tuple size_ * 8 + 1))`

Where *tuple size* is the size of a tuple in the page in bytes. The idea here is that each tuple requires one additional
bit of storage in the header. We compute the number of bits in a page (by mulitplying page size by 8), and divide this
quantity by the number of bits in a tuple (including this extra header bit) to get the number of tuples per page. The
floor operation rounds down to the nearest integer number of tuples (we don't want to store partial tuples on a page!)

Once we know the number of tuples per page, the number of bytes required to store the header is simply:

`headerBytes = ceiling(tupsPerPage/8)`

The ceiling operation rounds up to the nearest integer number of bytes (we never store less than a full byte of header
information.)

The low (least significant) bits of each byte represents the status of the slots that are earlier in the file. Hence,
the lowest bit of the first byte represents whether or not the first slot in the page is in use. The second lowest bit
of the first byte represents whether or not the second slot in the page is in use, and so on. Also, note that the
high-order bits of the last byte may not correspond to a slot that is actually in the file, since the number of slots
may not be a multiple of 8. Also note that all Java virtual machines
are big-endian.

Exercise 4

Implement the skeleton methods in:

- src/java/simpledb/storage/HeapPageId.java
- src/java/simpledb/storage/RecordId.java
- src/java/simpledb/storage/HeapPage.java

Although you will not use them directly in Lab 1, we ask you to implement `getNumEmptySlots()` and
`isSlotUsed()` in HeapPage. These require pushing around bits in the page header. You may find it helpful to look at
the other methods that have been provided in HeapPage or in `src/simpledb/HeapFileEncoder.java` to understand the
layout of pages.

You will also need to implement an Iterator over the tuples in the page, which may involve an auxiliary class or data
structure.

At this point, your code should pass the unit tests in HeapPageIdTest, RecordIDTest, and HeapPageReadTest.

After you have implemented `HeapPage`, you will write methods for `HeapFile` in this lab to calculate the
number of pages in a file and to read a page from the file. You will then be able to fetch tuples from a file stored on
disk.

Exercise 5

Implement the skeleton methods in:

- src/java/simpledb/storage/HeapFile.java

To read a page from disk, you will first need to calculate the correct offset in the file. Hint: you will need random
access to the file in order to read and write pages at arbitrary offsets. You should not call BufferPool methods when
reading a page from disk.

You will also need to implement the `HeapFile.iterator()` method, which should iterate through through the tuples of each page in the HeapFile. The iterator must use the `BufferPool.getPage()` method to access pages in the `HeapFile`. This method loads the page into the buffer pool and will eventually be used (in a later lab) to implement locking-based concurrency control and recovery. Do not load the entire table into memory on the open() call -- this will cause an out of memory error for very large tables.

At this point, your code should pass the unit tests in HeapFileReadTest.

2.6. Operators

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 `getNext` 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.

<!--
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 a tuple
containing the count of the number of affected tuples to the user-level
program.

-->

For this lab, you will only need to implement one SimpleDB operator.

Exercise 6.

Implement the skeleton methods in:

- src/java/simpledb/execution/SeqScan.java

This operator sequentially scans all of the tuples from the pages of the table specified by the `tableid` in the
constructor. This operator should access tuples through the `DbFile.iterator()` method.

At this point, you should be able to complete the ScanTest system test. Good work!

You will fill in other operators in subsequent labs.

2.7. A simple query

The purpose of this section is to illustrate how these various components are connected together to process a simple
query.

Suppose you have a data file, "some_data_file.txt", with the following contents:

```
1,1,1
2,2,2
3,4,4
```

You can convert this into a binary file that SimpleDB can query as follows:

```java -jar dist/simpledb.jar convert some_data_file.txt 3```

Here, the argument "3" tells conver that the input has 3 columns.

The following code implements a simple selection query over this file. This code is equivalent to the SQL statement `SELECT * FROM some_data_file`.

```
package simpledb;
import java.io.*;

public class test {

    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 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"), descriptor);
        Database.getCatalog().addTable(table1, "test");

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

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

}
```

The table we create 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. If this were a database server that was already running, we would have this catalog information loaded. We
need to load it explicitly to make this code self-contained.

Once we have finished 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, these operators are instantiated with references to the
appropriate table (in the case of `SeqScan`) or child operator (in the case of e.g. Filter). The test program then
repeatedly calls `hasNext` and `next` on the `SeqScan` operator. As tuples are output from the `SeqScan`, they are
printed out on the command line.

We **strongly recommend** you try this out as a fun end-to-end test that will help you get experience writing your own
test programs for simpledb. You should create the file "test.java" in the src/java/simpledb directory with the code above,
and you should add some "import" statement above the code,
and place the `some_data_file.dat` file in the top level directory. Then run:

```
ant
java -classpath dist/simpledb.jar simpledb.test
```

Note that `ant` compiles `test.java` and generates a new jarfile that contains it.

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. These may be minimal for Lab 1.
- 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 lab, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

This lab should be manageable for a single person, but if you prefer to work with a partner, this is also OK. Larger
groups are not allowed. Please indicate clearly who you worked with, if anyone, on your individual writeup.

3.2. Submitting your assignment

We will be using gradescope to autograde all programming assignments. You should have all been invited to the
class instance; if not, please check piazza for an invite code. If you are still having trouble, let us know and we can
help you set up. You may submit your code multiple times before the deadline; we will use the latest version as
determined by gradescope. Place the write-up in a file called lab1-writeup.txt with your submission.
You also need to explicitly add any other files you create, such as new *.java files.

The easiest way to submit to gradescope is with `.zip` files containing your code. On Linux/MacOS, you can
do so by running the following command:

```
$ zip -r submission.zip src/ lab1-writeup.txt
```

3.3. Submitting a bug

Please submit (friendly!) bug reports to 6.830-staff@mit.edu. 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.

If you are the first person to report a particular bug in the code, we will give you a candy bar!

3.4 Grading

75% 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, gradescope will replace your `build.xml` 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 also be careful changing our APIs. You should test that your code compiles the unmodified tests.

You should get immediate feedback and error outputs for failed tests (if any) from gradescope after submission. The score given will be your grade for the autograded portion of the assignment. An additional 25% of your grade will be based on the quality of your writeup and our subjective evaluation of your code. This part will also be published on gradescope after we finish grading your assignment.

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