**COMS2038 Lab 1: SimpleDB**

***Introduction***

In the lab assignments in COMS2038 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 that you may find useful in verifying that your code works.

Throughout the lab you will see numbered exercises, offset in boxes, that describe where you should write code and which unit tests you should expect to pass once you are done.

The remainder of this document describes the basic architecture of SimpleDB, and gives suggestions for coding, including the exercises you should complete.

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

***Quick jump to exercises:***

* [Section 1](http://ada.suda.edu.cn/courses/current/Lab/lab1.html#sec1) for Exercise 0
* [Section 2.2](http://ada.suda.edu.cn/courses/current/Lab/lab1.html#sec2.2) for Exercise 1
* [Section 2.3](http://ada.suda.edu.cn/courses/current/Lab/lab1.html#sec2.3) for Exercise 2
* [Section 2.4](http://ada.suda.edu.cn/courses/current/Lab/lab1.html#sec2.4) for Exercise 3
* [Section 2.5](http://ada.suda.edu.cn/courses/current/Lab/lab1.html#sec2.5) for Exercise 4 and 5
* [Section 2.6](http://ada.suda.edu.cn/courses/current/Lab/lab1.html#sec2.6) for Exercise 6

**1. Getting started**

Download the code from [cs133-lab1.tar.gz](http://ada.suda.edu.cn/courses/lab/cs133-lab1.tar.gz) and untar it.

To help you during development, we have provided a set of unit tests . These are by no means comprehensive, and you should not rely on them exclusively to verify the correctness of your lab.

**1.2. 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 should be obvious from the comments within and above each Java method where we think you should write code. You may need to add private methods and/or helper classes.

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 an order for the exercises in 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 fixed length 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.

**1.3. 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.

**1.4. Working in Eclipse**

[Eclipse](http://www.eclipse.org/) is a graphical software development environment that you might be more comfortable with working in. The instructions we provide were tested to work with Eclipse Mars, Juno, and Kepler with Java 1.6, 1.7, and 1.8

**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).
* On the file system, copy cs133-lab1.tar.gz to $W/cs133-lab1.tar.gz. Un-GZip and un-tar it, which will create a directory $W/cs133-lab1 (to do this, you can type tar -pzxvf cs133-lab1.tar.gz).
* In Eclipse, select File->New->Java Project.
* Enter "cs133-lab1" as the project name.
* Make sure the location is set to $W/cs133-lab1.
* Click finish, and you should be able to see "cs133-lab1" 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."

**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 "cs133-lab1" 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). Arguments such as "-Dtest=testname" can be specified in the "Main" Tab, "Arguments" textbox. Clicking the "Run" button should run the build targets and show you the results in Eclipse's console window.

**If right-clicking on build.xml doesn't work** you can also add an Ant view window. Go to Window -> Show view -> Ant. When the view opens, click the "add buildfile" icon. In the pop-up dialog box, select the build.xml under "cs133-lab1".

**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. 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.
* Indexes.

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 TupleDesc 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/simpledb/TupleDesc.java
* src/simpledb/Tuple.java

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

**Some helpful notes:**

* For many of the SimpleDb Java classes throughout the labs, you will need to add your own instance variables as needed.
* Remember that in Java, you check for equality using == only for primitive types. Any object, including String, should be compared using the equals() method.
* Standard Java classes that implement the interface java.lang.Iterable, such as java.util.ArrayList, can be useful for simple iterators like those required in TupleDesc and Tuple.
* In Type.java, the method getLen() returns the length in bytes.
* The Java keyword instanceof can be used to check if an object is an instance of a particular Java class.

**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/simpledb/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/simpledb/BufferPool.java

We have not provided unit tests for BufferPool. The functionality you implemented will be tested in the implementation of HeapFile below.

**Some helpful notes:**

* You should use the DbFile.readPage method to access pages of a DbFile. Think about how to get the correct DbFile!
* You won't use the arguments tid or perm until Lab 4.

**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.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. 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](http://en.wikipedia.org/wiki/Endianness).

**Exercise 4.** Implement the skeleton methods in:

* src/simpledb/HeapPageId.java
* src/simpledb/RecordID.java
* src/simpledb/HeapPage.java

Although you will not use them directly in Lab 1, we ask you to implement getNumEmptySlots() and isSlotFree() 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 insrc/simpledb/HeapFileEncoder.java to understand the layout of pages.

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

**Some helpful notes:**

* Be careful with rounding errors with Java int, as discussed in class.
* Implementing the iterator() method in HeapPage will likely be more complex than the iterator methods you have written so far. You will need to implement an Iterator for the existing tuples in the page, which can be done by creating a class that implements the interface java.util.Iterator. *Note: this is a different interface than java.lang.Iterable*.

Now that 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/simpledb/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 provide an Iterator of type DbFileIterator to 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 in the Iterator -- this will cause an out of memory error for very large tables!

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

**Some helpful notes:**

* You can use the global variable BufferPool.PAGE\_SIZE.
* The Java class java.io.RandomAccessFile may be useful for reading and writing to the on-disk file for a HeapFile.
* Pages in a HeapFile are numbered 0, 1, ... numPages()-1.
* You may find the method length() in java.io.File useful.

**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 query execution plan, or simply "plan", by passing lower-level operators into the constructors of higher-level operators, i.e., by 'chaining them together', forming a tree of operators. 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 this lab, you will only need to implement one SimpleDB operator; this operator is one of the access methods at the leaves of a query plan.

**Exercise 6.** Implement the skeleton methods in:

* src/simpledb/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 iterator() method provided by DbFile.

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

**Some helpful notes:**

* SeqScan implements the interface DbIterator, so be sure to read the comments in DbIterator.java. *Note: this is \*not\* the same as the DbFileIterator interface.*
* The work of this class should mostly be accomplished using the iterator for the appropriate HeapFile (think about how to find that DbFile!).
* You can use the TransactionId passed into the constructor to instantiate the HeapFile's iterator.

You will add other operators in subsequent labs.