198:437 198:539 Database Systems Implementation - Fall 2016

Programming Assignment 1 SimpleDB Heap Pages and Buffer Replacement Policies

Due Monday October 17, 11:59pm, via Sakai

Acknowledgement

This assignment is adapted from Prof. Sam Madden's 6.830 class at MIT and the cs544 class at U. Of Washington.

Clarifications

Please let us know if you encounter any problems with this assignment. We will post clarifications and fixes on Sakai as necessary.

Assignment goal

In this assignment, you will write parts of a basic database management system called SimpleDB.

SimpleDB is written in Java. We have provided you with a set of classes and interfaces. We have (sometimes partially) implemented some classes for you. You will need to write the code for some methods and classes. SimpleDB is designed as a complete system where each module can be implemented. In this assignment, we are only focussing on the HeapPage and BufferPool classes. We will grade your code by running a set of system tests. We have also provided a number of unit tests that you may find useful in verifying that your code works. Note 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.

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 assignment.

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

0. Find bugs, be patient

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 It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.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 assignment with an adventurous mindset. Don't get mad if something is not clear, or even wrong; rather, try to figure it out yourself, share your experience on the Sakai discussion board, or send us a friendly email. We will do our best to help with fixes as bugs are reported.

1. Getting started

These instructions are written for any Unix-based platform (e.g., Linux, MacOS, etc.). Because the code is written in Java, it should work under Windows as well, though the

directions in this document may not apply.
Download the code from Sakai and untar it. In linux:

```
$ tar zxvf simpledb_437_hws_F16.tar.gz
$ cd simpledb_437_hws_F16
```

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.
For more details about how to use Ant, see the manual. The Running Ant section provides details about using the ant command. However, the quick reference table below should be sufficient for working on the assignments.

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/si`
`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`.

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

```
$ cd simpledb_437_hw_F16
$ # run a specific unit test
$ ant runtest -Dtest=TupleTest
```

You should see output similar to:

```
Buildfile:/[your_path]/build.xml

compile:

testcompile:

runtest:
  [junit] Running simpledb.TupleTest
  [junit] Testsuite: simpledb.TupleTest
  [junit] Tests run: 3, Failures: 0, Errors: 0, Time
elapsed: 0.036 sec
  [junit] Tests run: 3, Failures: 0, Errors: 0, Time
elapsed: 0.036 sec
  [junit]
  [junit] Testcase: modifyFields took 0.08 sec
  [junit] Testcase: getTupleDesc took 0 sec
  [junit] Testcase: modifyRedcordID took 0.001 sec

BUILD SUCCESSFUL
Total time: 1 second
```

The output above indicates that the code passed the TupleTest.

Overview

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 some pieces of code that is not implemented. You may need to add private methods and/or helper classes. You may change APIs, but make sure our tests still run and make sure to mention, explain, and defend your decisions in your writeup. **There are three main parts A,B,C that you should implement in order. They are highlighted in red in this document.**

In addition to the methods that you need to fill out for this assignment, the class interfaces contain numerous methods that you need not implement in this assignment. We have already implemented the class to manage tuples namely Tuple, TupleDesc, as well as Field, IntField, StringField, Type, Catalog, HeapPageId, HeapFile,and SeqScan for you. Note that you only need to support integer and (fixed length) string fields and fixed length tuples.

You will need to :

- Check that the code you install passes the following tests:
  - TupleTest
  - TupleDescTest
  - CatalogTest
  - RecordIDTest
  - HeapPageIdTest
- 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. (You may not have to modify HeapFile)

Transactions, locking, and recovery

As you look through the interfaces that we have provided you, you will see a number of references to locking, transactions, and recovery. You do not need to support these features. 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 assignment); 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:

- Views.
- Data types except integers and fixed length strings.
- Query optimizer.
- Indices.

In the rest of this Section, we describe each of the main components of SimpleDB that you will need to implement in this assignment. 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.

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 assignment. 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.We have already implemented the classes:

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

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.

To learn more about catalogs in a DBMS, take a look at Section 12.1 of the R&G book.

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()).We have already implemented the class:

src/simpledb/Catalog.java

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. 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 assignment we will not be implementing an eviction policy. If more than numPages requests are made for different pages, then instead of implementing an eviction policy, you may throw a DbException.
The Database class provides a static method, Database.getBufferPool(), that returns a reference to the single BufferPool instance for the entire SimpleDB process.

A. You will need to implement the getPage() method in:

src/simpledb/BufferPool.java

Details on the functionality you will need to implement will be given below in Part C. As a first step, you can implement a very basic version of that loads a page in any buffer frame if the page is not already in the buffer. To load a page,you should use the **DbFile.readPage** method to access pages of a DbFile.
Note: You are required to implement BufferPool.getPage() to be able to run the tests in Part B. The implementation at this point can be very primitive (e.g., always load the page in frame 0). You should not submit this implementation if you implement Part C.

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), plus 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 32-bit integers representing the header (headersize), followed by the BufferPool.PAGE_SIZE - headersize bytes of actual page content. The number of 32-bit integers in the header (headersize) is defined by the formula:
int tuplesPerPage = (BufferPool.PAGE_SIZE*8) / ((_td.getSize()*8)+1);
int headersize = (int)Math.ceil(tuplesPerPage/8);
Where *_td.getSize()* provides the size of a tuple in the page in bytes.
The header is stored as set of integer, which represents bitmap data. To access the header bit for a slot you then need to perform bit manipulation. The low (least significant) bits of each integer represent the status of the slots that are earlier in the file. Hence, the lowest bit of the first integer represents whether or not the first slot in the page is in use. Also, note that the high-order bits of the last such integer may not correspond to a slot that is actually in the file, since the number of slots may not be a multiple of 32. Also note that all Java virtual machines are big-endian.
The page content of each page consists of floor(BufferPool.PAGE_SIZE*8) / ((_td.getSize()*8)+1) tuple slots, where the 0-indexed *i*th slot begins *i * tuple size* bytes into the page.

B. You will need to implement the getNumEmptySlots(), setSlot() and getSlot() methods in:

src/simpledb/HeapPage.java

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 and src/simpledb/HeapFile.java to understand the layout of pages.

At this point, your code should pass the unit tests in HeapFileReadTest, and HeapPageReadTest. (You need a working --but not necessarily good-- implementation of BufferPool.getPage() to pass these tests).

2.6. Buffer Pool Eviction Strategy

You need to implement a BufferPool eviction strategy. Unlike the implementation required in Section 2.4, which let you ignore the limit on the maximum number of pages in the buffer pool defined by the constructor argument numPages, you will choose a page eviction policy and instrument any previous code that reads or creates pages to implement your policy.
C. You will need to re-implement the getPage(), pinPage(), unpinPage(), and evictPage() methods in:

src/simpledb/BufferPool.java

You may need to update more functions and class members. In particular you implementation should have the following:

- the buffer pool should be represented as an array of Page: Page buffer[]
- if the method unpinPage() is called when the pin_count == 0, it should throws DbException
- if the method unpinPage() is called with dirty_page == true, the method flushPage() should be called.

When more than numPages pages are in the buffer pool, one page should be evicted from the pool before the next is loaded.
You will have to implement the two following eviction policies:

- MRU
- LRU

Note that we do not require to implement disk operations, so you do not need to worry about writing dirty pages back to disk! All you have to do for this question is to implement the evictPage() method. You may also need to modify your getPage() method to make sure to keep track of frame usage, pincount and dirty bits, as well other methods but you do NOT need to implement flushAllPages() nor flushPage().
At this point your implementation should pass the test systemtest.PinCountEvictionTest. Note that this test does not test for correctness of your eviction strategies, but checks whether you are staying within the BufferPool memory allocation. In addition, you should use the tests systemtest.PinCountTest and systemtest.BPPinCountTest to test your implementations of MRU and LRU. Sample results are available in Sakai. Your code is expected to return the same number of hits and misses. (Hint! Looking at the test sources should help you understand how the policy is set.)

Submission Instructions

You should submit your assigment via Sakai.
You must submit your code (see below) as well as a short (1 page, maximum) writeup describing your implementation. This writeup should:

- Describe any design decisions you made. For example any class or complex data structure you add to the project.
- 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.

To submit your code, please create a [YourName]_prog1.tar.gz tarball (such that, untarred, it creates a [YourName]_prog1/src/simpledb directory with your code) and submit it to Sakai.
You may submit your code multiple times; we will use the latest version you submit that arrives before the deadline. Please also include your writeup as a PDF or text file in the submission package.