6.830 Problem Set 2: SimpleDB

Assigned: Thursday, September 22
Due: Thursday, October 6

In the remaining problem sets in 6.830, you will write a basic database management system called SimpleDB. For this problem set, you will focus on implementing the core modules required to process simple queries; in future problem sets, you will add support for transactions, locking, concurrent queries, indices, and optimization.

SimpleDB is written in Java. We have provided you with a set of (mostly uncoded) classes and interfaces. You will need to write the code for these classes. We will grade your problem set by running it against a collection of test programs, some of which we have given to you so that you can test your own software.

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 problem set.

There are also a few short questions (unrelated to SimpleDB) in Section 4 that you should answer. These questions are meant to give you practice designing databases and thinking about database performance.

We **strongly recommend** that you start as early as possible on the SimpleDB portion of this problem set. It requires you to write a fair amount of code!

News

09/25/05, 2:30pm: We handed out the first candybar; there were two bugs which affected `PageEncoder.java` and `HeapPage.java`. A new tarball for problem set 2 is up for download. If you hadn't yet started working on the problem set, you don't have to worry about this. Otherwise, carefully read the explanation below.

First of all, `PageEncoder.java` would leave empty pages at the end of the heapfile. We modified it to not flush empty pages to disk, except for the case where conversion would otherwise lead to an empty file. Secondly, the number of bytes in a page computed by `HeapPage.createEmptyPageData()` was wrong.

If you already started working on the problem set, we suggest you replace the old `PageEncoder.java` with the version from the new tarball. In addition, you should make sure that `createEmptyPageData()` in `HeapPage.java` looks like this:

```
  public static byte[] createEmptyPageData(int tableid) {
    TupleDesc td = Catalog.Instance().getTupleDesc(tableid);
    int hb = (((BufferPool.PAGE_SIZE / td.getSize()) / 32) +1) * 4;
    int len = BufferPool.PAGE_SIZE + hb;
    return new byte[len]; //all 0
  }
```

0. Find bugs, be patient, earn candybars

This is the first time we have used SimpleDB in 6.830. Consequently, it is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc. We apologize profusely. We did our best, but, alas, we are fallible human beings.

We ask you, therefore, to do this problem set 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 sending bugfixes, new tarballs, etc.

...and if you find a bug in our code, we'll give you a candybar (see Section 3.3)!

1. 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 problem set); 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:

- 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 1.6).
- Support for views.
- Support for any data types except integers.
- Support for locking, query optimization, or indices, *for this problem set only*. You will add support for these in future problem sets.

In the rest of this Section, we describe each of the main components of SimpleDB. You should use 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.

You may also wish to consult the JavaDoc for SimpleDB.

1.1. Fields and Tuples

Tuples in SimpleDB are extremely 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. For the purposes of this assignment, the only field type you need to code is `IntField`. `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 an collection of `Type` objects, one per field in the tuple, each of which describes the type of the corresponding field.

1.2. 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 (defined by the constant `BufferPool.PAGE_SIZE`). 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, followed by the `BufferPool.PAGE_SIZE` bytes of actual page content. The number of 32-bit integers in the header is defined by the formula:

```
((BufferPool.PAGE_SIZE / tuple size) / 32 ) +1 )
```

Where *tuple size* is the size of a tuple in the page in bytes.

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/`*`tuple size`*`)` tuple slots, where the 0-indexed *i*th slot begins `i * `*`tuple size`* bytes into the page.

You may find it helpful to look at `src/simpledb/PageEncoder.java` or `src/simpledb/HeapPage.java` to understand the layout of pages.

1.3. 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 problem set, you will need to implement the following SimpleDB operators:

- *SeqScan*: This operator sequentially scans all of the tuples from the pages of the table specified as the `tableid` in the constructor. This operator should access pages through the `BufferPool.getPage()` method.
- *Filter*: This operator only passes on tuples that pass a `Predicate` that is specified as part of its constructor.
- *Join*: This operator joins tuples from its two children according to JoinPredicate that is passed in as part of its constructor. You should implement a simple nested loops join.
- *Aggregate*: This operator implements basic SQL aggregates with `GROUP BY`. You should implement the five SQL aggregates (`COUNT, SUM, AVG, MIN, MAX`) and support grouping. You only need to support aggregates over a single field, and grouping by a single field. You will probably want to use a Java `HashMap` for grouping. You do not need to worry about the situation where the number of groups exceeds available memory.
- *Insert*: This operator adds the tuples it reads from its child operator to the `tableid` specified in its constructor. It should use the `BufferPool.insertTuple()` method to do this.

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

1.4. 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 BufferPool. It consists of a fixed number of pages, defined by the constant `NUM_PAGES`. When more than this many pages are in the buffer pool, one page should be evicted from the pool before the next is loaded. The choice of eviction policy is up to you; it is not necessary to do something sophisticated.

`BufferPool` provides a static method, `BufferPool.Instance()` that provides a reference to a single instance of BufferPool that is allocated for the entire SimpleDB process.

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.

1.5. 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.

Similar to `BufferPool`, `Catalog` provides a method, `Catalog.Instance()` that provides a reference to a single instance of Catalog that is allocated for the entire SimpleDB process.

1.6. A simple query

The purpose of this section is to illustrate how these various components are connected together to process a simple query. The code for this Section is taken from `src/simpledb/test/DemoTest.java`.

The following code implements a simple selection query over a data file consisting of three columns of integers. (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 WHERE field0 > 5`. This code is the body of a test routine similar other tests in the `test` directory; we have omitted some of the header information from this file for brevity.

```
  // construct a 3-column table schema
  Type typeAr[] = new Type[3];
  typeAr[0] = Type.INT_TYPE;
  typeAr[1] = Type.INT_TYPE;
  typeAr[2] = Type.INT_TYPE;
  TupleDesc t = new TupleDesc(typeAr);

  // 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"));
  Catalog.Instance().addTable(table1, t);

  // construct the query: SeqScan spoonfeeds tuples to Filter, which
uses
  // a predicate that filters tuples based on the first column: only
fields
  // greater than 5 are passed up.
  TransactionId tid = new TransactionId();
  SeqScan ss = new SeqScan(tid, table1.id());
  Predicate p = new Predicate(0, Predicate.GREATER_THAN, new
IntField(new Integer(5)));
  Filter f = new Filter(p, ss);

  // and run it
  try {
    f.open();

    try {
      while (true) {
        Tuple tup = f.getNext();
        tup.print();
      }
    } catch(NoSuchElementException e) {
      f.close();
    }
  } catch (TransactionAbortedException e) {
    e.printStackTrace();

  } catch (DbException e) {
    e.printStackTrace();
    return false;
  }

  BufferPool.Instance().transactionComplete(tid);
```

The table we create has three integer fields. To express this, we create a `TupleDesc` object and pass it an array of `Type` objects. 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 this only for the purposes of this test.)

Once we have finished initializing the database system, we create a query plan. Our plan consists of two operators: a `SeqScan` operator that scans the tuples from disk, and a

`Filter` operator that applies a specific `Predicate` to tuples that are input to it. These operators are instantiated with references to the appropriate table (in the case of SeqScan) or child operator (in the case of Filter). The test program then repeatedly calls `getNext` on the `Filter` operator, which will in turn call `getNext` on its child, the `SeqScan` operator. As tuples are output from the `Filter`, they are printed out on the command line.

2. Getting started

These instructions are written for MIT server 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, though the build tools and other scripts we have provided will not run (you should be able to get the code to compile with a graphical IDE such as Eclipse.)

Download the code from `6.830-ps2.tar.gz` and untar it. For example:

```
$ wget 6.830-ps2.tar.gz
$ gunzip 6.830-ps2.tar.gz
$ tar xvf 6.830-ps2.tar
$ cd 6.830-ps2
[on MIT server: add sipb]
[on MIT server: setenv JAVACMD `which java`]
$ ant
```

SimpleDB uses the `ant` build tool to compile and run the code. Ant works in a very similar way 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 MIT server, `ant` is included in the `sipb` locker, which you can get to by typing `add sipb` at the MIT server prompt. Note that on some versions of MIT server you must also set the `JAVACMD` environment variable (as in the above example) for `ant` to be able to find the correct java compiler.

You will need to fill in the routines in the code modules in the `src/simpledb/` directory. You can compile the code by typing `ant` in the `6.830-ps2` directory.

The simplest test you can run is `DemoTest` from above. Create a file `some_data_file.txt` and use `PageEncoder` to convert it to `some_data_file.dat`. Then run the test. For example:

```
$ wget http://db.csail.mit.edu/6.830/some_data_file.txt
$ java -jar dist/simpledb.jar convert some_data_file.txt 3
$ java -jar dist/simpledb.jar simpledb.test.DemoTest
7       8       9
10      11      12
```

(Notice that, when you run this using the tarball we have provided, the test fails with a `NullPointerException`. You need to write some code before this test will run!)

2.1 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.

You can also add tests to the `simpledb/src/simpledb/test/` directory, in the same format as the `DemoTest.java` code shown above.

2.2. Implementation hints

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 programs still run and make sure to mention, explain, and defend your decisions in your writeup.)

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

- Implement the classes to manage tuples, namely `Tuple`, `TupleDesc`, `Field`, and `IntField` We already implemented `Type` for you. Since you only need to support integer fields and fixed length tuples, these are straightforward.
- Implement the access methods, `HeapPage` and `HeapFile`. A good portion of these files has already been written for you. You may wish to defer the implementation of the methods that add and delete records from pages until later.
- Implement the `BufferPool` constructor and the `getPage()` method. Eventually, you will need to implement the insert and delete methods, as well as the code to evict pages, but these can be deferred until later.
- Implement the operator `SeqScan`.
- Implement the `Catalog` (this should be very simple).
- At this point, you should be able to pass test1 of `grader.pl`.
- Implement the additional operators: `Join`, `Filter`, and `Aggregate` and verify that the additional tests corresponding to these operators work. Be sure to read the comments in the fields for these operators to understand how they should work.
- Implement the `Insert` and `Delete` operators and the methods in `BufferPool`, `HeapFile`, and `HeapPage` that support them. Test to see that the test files for inserting and deleting tuples work properly. Note that SimpleDB does not implement any kind of consistency or integrity checking, so it is possible to insert

duplicate records into a file and there is no way to enforce primary or foreign key constraints.
- Complete the implementation by adding support for evicting pages from the buffer pool.

2.3. Transactions and locking

As you look through the interfaces we have provided you, you will see a number of references to locking and transactions. You do not need to support locking in this problem set, but you should keep these parameters in the interfaces of your code because you will be implementing transactions and locking in a future problem set. 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.

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, including your choice of page eviction policy, join operator implementation, aggregate operator implementation, etc.
- Discuss and justify any changes you made to the API.

3.1. Collaboration

This problem set 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 writeup.

3.2. Submitting your assignment

To submit your code, please create a `6.830-ps2.tar.gz` tarball (such that, untarred, it creates a `6.830-ps2/simpledb/src/` directory with your code) and email it. You may submit your code multiple times; we will use the
latest version you send that arrives before the submission deadline (before class on the due date). If applicable, please indicate your partner in your email. Please also attach your writeup as a PDF or text file.

3.3. Submitting a bug

Please submit (friendly!) bug reports. When you do, please try to include:

- A description of the bug.

- A `.java` file we can drop in the `src/simpledb/test` 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 `PageEncoder`.

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

3.4 Grading

50% of your grade will be based on whether or not your code passes the test program (`grader.pl`) we run over it. These programs will be a superset of the test programs we have provided for you. Before handing in your code, you should make sure you pass all tests in `grader.pl`. Notice that `grader.pl` gives you some help on its usage:

```
$ ./grader.pl -h

grader.pl -- SimpleDB torture

Usage: grader.pl [-c] [-h] [-t TEST] [-v] [-x]
  -c            : don't clear .txt and .dat files on exit, default off
  -h            : print this help
  -t TEST       : only run test TEST, default all tests
  -v            : increase verbosity, default off
  -x            : exit on failed test, default off

Tests
  1 : simple table dump
  2 : filters
  3 : delete
  4 : insert
  5 : join
  6 : aggregates
```

You should look through the source code of `grader.pl` and make sure you understand roughly what is going on. It might help you debug your code.

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

```
$ gunzip 6.830-ps2.tar.gz
$ tar xvf 6.830-ps2.tar
$ cd ./6.830-ps2
[replace grader.pl, PageEncoder.java, and test/]
$ ant
$ ./grader.pl
[additional tests]
```

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

An additional 35% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

The final 15% of your grade will be based on the additional questions in Section 4.

4. Additional questions

Please answer additional questions in this section (which are not related to SimpleDB). For these questions, suppose that Dana Bass wants to create a relational database that stores information about sales records of her clothing company for pets. Specifically, this database stores information about:

- (Human) Customers, including their name, shipping address, and credit card number
- Animals, including their species, breed, clothing size, and owner
- Orders, including the date they were placed, the customer who placed the order, and the shipping date of the order
- Products, including their type (e.g., shirt, hat), the species of animal they are for, their color, their price, and the number in stock
- The quantity of each product included in an order

1. Propose a relational schema for this database. Indicate the primary and foreign keys. Propose a reasonable set of functional dependencies for this schema. Show that your schema is in 3NF or BCNF.

2. Show the SQL for a query that computes all orders that are for a species of pet that the customer placing the order does not own.

For the following questions, suppose there are 10,000 customers, 100,000 orders, 20,000 pets, 1,000 products, and 200,000 total products purchaes. You may also assume that each row of each table occupies 100 bytes, each disk page is 1,000 bytes, and there is sufficient memory for any intermediate data structures (e.g., hash tables, intermediate join results) to be memory resident. You may assume a uniform distribution of pets to customers, order to customers, products to orders, etc.

3. Propose a set of index structures for this query that you believe would be a good choice for making it run efficiently.

4. Show an efficient equivalent relational algebra for this plan, in a tree structured representation. For each node in the tree, choose a particular implementation of the operator in question. Indicate which indices are used where as well as the use of SARGable predicates as needed.

5. Estimate the number of random and sequential disk I/Os that would be required to compute the result of this query plan. Show your work.

6. Suppose there is only 1MB of memory available for intermediate data structures. Would this affect your plan choice? Re-estimate the number of sequential and random I/Os required.

7. How many hours did you spend writing SimpleDB?