Search this site

Homeworks >

Project 1

CS186 Project 1: SimpleDB

CS186, UC Berkeley, Fall 2013
Points: 5% of your final grade
Note: This project is to be done alone or in pairs!
Assigned: Wednesday, September 11
Due: Wednesday, September 25, 11:59PM (You have 4 slip days for all CS186 projects, use them wisely)

In the project assignments in CS186, you will write a basic database management system called SimpleDB. For this project, you will focus on implementing the core modules required to access stored data on disk; in future projects, 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.

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 project.

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

1. Getting started

These instructions are written for Linux. We would like everyone to run Linux on a virtual machine, which will also be used for future projects. This will facilitate with grading and fixing bugs. Instructions to setup the VM are below.

Development Environment

These are the steps to get your development environment set up.

Virtual Machine

For this assignment, you will be using a virtual machine to develop and run the code. Follow the instructions at http://beta.saasbook.info/bookware-vm-instructions to install VirtualBox, and the virtual machine image. The virtual machine image already includes most of the software necessary to run the code. We will install extra packages below.

Note: After opening a terminal on the VM, if the language is incorrect, run `sudo dpkg-reconfigure keyboard-configuration` **and choose English(US).**

We HIGHLY, HIGHLY recommend using the virtual machine for this assignment. Please try as hard as possible to use the VM.

Getting the Base Code

In a fresh virtual machine, open up the terminal. First, install the following packages.

```
$ sudo apt-get update
$ sudo apt-get install openjdk-6-jdk
$ sudo apt-get install ant
```

Next, download the code from https://www.dropbox.com/s/qs4uxgwvsc2m7bj/CS186-proj1.tar.gz and untar it. For example:

```
$ wget https://www.dropbox.com/s/qs4uxgwvsc2m7bj/CS186-proj1.tar.gz
$ tar xvzf CS186-proj1.tar.gz
$ cd CS186-proj1
```

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.

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 CS186-proj1
```

```
$ # 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 project, 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. The Running Ant section provides details about using the `ant` command. However, the quick reference table below should be sufficient for working on the projects.

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`.
`ant handin`	Generate tarball for submission.

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. 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 project, the class interfaces contain numerous methods that you need not implement until subsequent projects. These will either be indicated per class:

```
// Not necessary for proj1.
public class Insert implements DbIterator {
```

or per method:

```
public boolean deleteTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for proj1
    return false;
}
```

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

We suggest an ordering of implementations throughout the document, 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 project.

  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 project, but you should keep these parameters in the interfaces of your code because you will be implementing transactions and locking in a future project. 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 project); 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 project), 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 projects.
- Views.
- Data types except integers and fixed length strings.
- (In this project) 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. Note that for Project 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 projects.

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 project. 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/TupleDesc.java`
- `src/java/simpledb/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/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 projects, you will implement an eviction policy. For this project, 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 project, if more than numPages requests are made for different pages, then instead of implementing an eviction policy, you may throw a DbException. In future projects 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/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.

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 `BufferPool.PAGE_SIZE - # header bytes` 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:

```
tupsPerPage = floor((BufferPool.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.

Exercise 4. Implement the skeleton methods in:

- `src/java/simpledb/HeapPageId.java`
- `src/java/simpledb/RecordID.java`
- `src/java/simpledb/HeapPage.java`

Although you will not use them directly in Project 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 in `src/java/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 project 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/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 project) 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 this project, you will only need to implement one SimpleDB operator.

Exercise 6. Implement the skeleton methods in:

- `src/java/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 `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 projects.

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 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 (1 page, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made. These may be minimal for Project 1.
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe briefly who worked on what (if you worked with a partner).
- Describe how long you (and your partner) spent on the project, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

This assignment can be completed alone or with a partner.

3.2. Submitting your assignment

The files you need to submit are:

1. `CS186-proj1.tar.gz` tarball (such that, untarred, it creates a `CS186-proj1/src/java/simpledb` directory with your code). You can use the `ant handin` target to generate the tarball.

2. `answers.txt` containing the write-up.

3. `README` -- If you are using slip days, your `README` should contain a single digit indicating the number of slip days you wish to use. For example, if you wanted to use two slips days, the `README` would consist of only one line with the number: `2`. If you are not using slip days, the `README` should be empty.

4. `MY.PARTNERS` -- This is auto-generated for you when you run the `submit proj1` script.

You may submit your code multiple times; we will use the latest version you submit that arrives before the deadline (before 11:59 PM on the due date).

Note: The simplest way to create your submission is to create a directory on the inst machines, with just the files above.

On an inst machine, run `submit proj1`:

```
% submit proj1
Please enter the logins of your partner(s), if any.
Enter '.' to stop.
...
...
Created MY.PARTNERS file.
Looking for files to turn in....
Submitting answers.txt.
Submitting CS186-proj1.tar.gz.
Submitting README.
The files you have submitted are:
./MY.PARTNERS ./README ./CS186-proj1.tar.gz ./answers.txt
Is this correct? [yes/no] y
Copying submission of assignment proj1....
Submission complete.
```

3.3. Submitting a bug

Please submit (friendly!) bug reports by emailing `cs186projbugs@gmail.com`. 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, we 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. 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:

```
$ tar xvzf CS186-proj1.tar.gz
$ cd ./CS186-proj1
[replace build.xml and test]
$ ant test
$ ant systemtest
[additional tests]
```

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

An additional 25% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

We expect that the work you submit will be solely your (and your partner's) work. We will be running the standard code duplication checking software.

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

Acknowledgements

Thanks to our friends and colleagues at MIT and UWashington for doing all the heavy lifting on creating SimpleDB!

Comments

You do not have permission to add comments.