

6.830 Lab 2: SimpleDB Operators

Assigned: September 30, 2010

Due: Friday, October 15, 2010

Version History:

- 9/30/10 : Initial version
- 10/5/10 : Updated due date

In this lab assignment, you will write a set of operators for SimpleDB to implement table modifications (e.g., insert and delete records), selections, joins, and aggregates. These will build on top of the foundation that you wrote in Lab 1 to provide you with a database system that can perform simple queries over multiple tables.

Additionally, we ignored the issue of buffer pool management in Lab 1; we have not dealt with the problem that arises when we reference more pages than we can fit in memory over the lifetime of the database. In Lab 2, you will design an eviction policy to flush stale pages from the buffer pool.

You do not need to implement transactions or locking in this lab.

The remainder of this document gives some suggestions about how to start coding, describes a set of exercises to help you work through the lab, and discusses how to hand in your code. This lab requires you to write a fair amount of code, so we encourage you to start early!

0. Find bugs, be patient, earn candy bars

SimpleDB is a relatively complex piece of code. It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.

We ask you, therefore, to do this lab with an adventurous mindset. Don't get mad if something is not clear, or even wrong; rather, try to figure it out yourself or send us a friendly email. We promise to help out by posting bug fixes, new tarballs, etc., as bugs and issues are reported.

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

1. Getting started

You should begin with the code you submitted for Lab 1 (if you did not submit code for Lab 1, or your solution didn't work properly, contact us to discuss options.) We have provided you with extra test cases for this lab that are not in the original code distribution you received. We reiterate 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.

You will need to add these new test cases to your release. The easiest way to do this is to untar the new code in the same directory as your top-level simpledb directory, as follows:

- Make a backup of your Lab 1 solution by typing :

```
$ tar -cvzf 6.830-lab1-submitted.tar.gz 6.830-lab1
```

- Change to the directory that contains your top-level simpledb code:

```
$ cd 6.830-lab2
```

- Download the new tests and skeleton code for Lab 2 from <http://db.csail.mit.edu/6.830/6.830-lab2-supplement>.

tar.gz:

```
$ wget http://db.csail.mit.edu/6.830/6.830-lab2-supplement.tar.gz
```

- Extract the new files for Lab 2 by typing:

```
tar -xvzf 6.830-lab2-supplement.tar.gz
```

- Eclipse users will have to take one more step for their code to compile. (First, make sure the project is in your workspace in Eclipse following similar steps as done in lab1.) Under the package explorer, right click the project name (probably 6.830-lab1), and select Properties. Choose Java Build Path on the left-hand-side, and click on the Libraries tab on the right-hand-side. Push the Add JARs... button, select zql.jar and jline-0.9.94.jar, and push OK, followed by OK. Your code should now compile.

1.3. Implementation hints

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

We suggest exercises along this document to guide your implementation, but you may find that a different order makes more sense for you. As before, we will grade your assignment by looking at your code and verifying that you have passed the test for the ant targets `test` and `systemtest`. See Section 3.4 for a complete discussion of grading and list of the tests you will need to pass.

Here's a rough outline of one way you might proceed with your SimpleDB implementation; more details on the steps in this outline, including exercises, are given in Section 2 below.

- Implement the operators `Filter` and `Join` and verify that their corresponding tests work. The Javadoc comments for these operators contain details about how they should work. We have given you implementations of `Project` and `OrderBy` which may help you understand how other operators work.
- Implement `IntAggregator` and `StringAggregator`. Here, you will write the logic that actually computes an aggregate over a particular field across multiple groups in a sequence of input tuples. Use integer division for computing the average, since SimpleDB only supports integers. `StringAggregator` only needs to support the `COUNT` aggregate, since the other operations do not make sense for strings.
- Implement the `Aggregate` operator. As with other operators, aggregates implement the `DbIterator` interface so that they can be placed in SimpleDB query plans. Note that the output of an `Aggregate` operator is an aggregate value of an entire group for each call to `next()`, and that the aggregate constructor takes the aggregation and grouping fields.
- Implement the methods related to tuple insertion, deletion, and page eviction in `BufferPool`. You do not need to worry about transactions at this point.
- Implement the `Insert` and `Delete` operators. Like all operators, `Insert` and `Delete` implement `DbIterator`, accepting a stream of tuples to insert or delete and outputting a single tuple with an integer field that indicates the number of tuples inserted or deleted. These operators will need to call the appropriate methods in `BufferPool` that actually modify the pages on disk. Check that the tests 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.

At this point you should be able to pass all of the tests in the ant `systemtest` target, which is the goal of this lab.

You'll also be able to use the provided SQL parser to run SQL queries against your database! See [Section 2.7](#) for a brief tutorial and a description of an optional contest to see who can write the fastest SimpleDB implementation.

Finally, you might notice that the iterators in this lab extend the `AbstractDbIterator` class instead of implementing the `DbIterator` interface. Because the implementation of `next/hasNext` is often repetitive, annoying, and error-prone, `AbstractDbIterator` implements this logic generically, and only requires that you implement a

simpler `readNext`. Feel free to use this style of implementation, or just implement the `DbIterator` interface if you prefer. To implement the `DbIterator` interface, remove `extends AbstractDbIterator` from iterator classes, and in its place put `implements DbIterator`.

2. SimpleDB Architecture and Implementation Guide

2.1. Filter and Join

Recall that SimpleDB `DbIterator` classes implement the operations of the relational algebra. You will now implement two operators that will enable you to perform queries that are slightly more interesting than a table scan.

- *Filter*: This operator only returns tuples that satisfy a `Predicate` that is specified as part of its constructor. Hence, it filters out any tuples that do not match the predicate.
- *Join*: This operator joins tuples from its two children according to a `JoinPredicate` that is passed in as part of its constructor. We only require a simple nested loops join, but you may explore more interesting join implementations. Describe your implementation in your lab writeup.

Exercise 1. Implement the skeleton methods in:

- `src/simplydb/Predicate.java`
- `src/simplydb/JoinPredicate.java`
- `src/simplydb/Filter.java`
- `src/simplydb/Join.java`

At this point, your code should pass the unit tests in `PredicateTest`, `JoinPredicateTest`, `FilterTest`, and `JoinTest`. Furthermore, you should be able to pass the system tests `FilterTest` and `JoinTest`.

2.2. Aggregates

An additional SimpleDB operator implements basic SQL aggregates with a `GROUP BY` clause. 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.

In order to calculate aggregates, we use an `Aggregator` interface which merges a new tuple into the existing calculation of an aggregate. The `Aggregator` is told during construction what operation it should use for aggregation. Subsequently, the client code should call `Aggregator.merge()` for every tuple in the child iterator. After all tuples have been merged, the client can retrieve a `DbIterator` of aggregation results. Each tuple in the result is a pair of the form `(groupValue, aggregateValue)`, unless the value of the group by field was `Aggregator.NO_GROUPING`, in which case the result is a single tuple of the form `(aggregateValue)`.

Note that this implementation requires space linear in the number of distinct groups. For the purposes of this lab, you do not need to worry about the situation where the number of groups exceeds available memory.

Exercise 2. Implement the skeleton methods in:

- `src/simplydb/IntAggregator.java`
- `src/simplydb/StringAggregator.java`
- `src/simplydb/Aggregate.java`

At this point, your code should pass the unit tests `IntAggregatorTest`, `StringAggregatorTest`, and `AggregateTest`. Furthermore, you should be able to pass the `AggregateTest` system test.

2.3. HeapFile Mutability

Now, we will begin to implement methods to support modifying tables. We begin at the level of individual pages and files. There are two main sets of operations: adding tuples and removing tuples.

Removing tuples: To remove a tuple, you will need to implement `deleteTuple`. Tuples contain `RecordIDs` which allow you to find the page they reside on, so this should be as simple as locating the page a tuple belongs to and modifying the headers of the page appropriately.

Adding tuples: The `addTuple` method in `HeapFile.java` is responsible for adding a tuple to a heap file. To add a new tuple to a `HeapFile`, you will have to find a page with an empty slot. If no such pages exist in the `HeapFile`, you need to create a new page and append it to the physical file on disk. You will need to ensure that the `RecordID` in the tuple is updated correctly.

Exercise 3. Implement the remaining skeleton methods in:

- `src/simpliedb/HeapPage.java`
- `src/simpliedb/HeapFile.java` (note that you do not necessarily need to implement `writePage` at this point.)

To implement `HeapPage`, you will need to modify the header bitmap for methods such as `addTuple()` and `deleteTuple()`. You may find that the `getNumEmptySlots()` and `getSlot()` methods we asked you to implement in Lab 1 serve as useful abstractions. Note that there is a `setSlot` method provided as an abstraction to modify the filled or cleared status of a tuple in the page header.

Note that it is important that the `HeapFile.addTuple()` and `HeapFile.deleteTuple()` methods access pages using the `BufferPool.getPage()` method; otherwise, your implementation of transactions in the next lab will not work properly.

Implement the following skeleton methods in `src/simpliedb/BufferPool.java`:

- `insertTuple()`
- `deleteTuple()`

These methods should call the appropriate methods in the `HeapFile` that belong to the table being modified (this extra level of indirection is needed to support other types of files — like indices — in the future).

At this point, your code should pass the unit tests in `HeapPageWriteTest` and `HeapFileWriteTest`. We have not provided additional unit tests for `HeapFile.deleteTuple()` or `BufferPool`.

2.4. Insertion and deletion

Now that you have written all of the `HeapFile` machinery to add and remove tuples, you will implement the `Insert` and `Delete` operators.

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 the number of affected tuples. This is implemented by returning a single tuple with one integer field, containing the count.

- *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.

Exercise 4. Implement the skeleton methods in:

- `src/simpliedb/Insert.java`
- `src/simpliedb/Delete.java`

At this point, your code should pass the unit tests in `InsertTest`. We have not provided unit tests for `Delete`. Furthermore, you should be able to pass the `InsertTest` and `DeleteTest` system tests.

2.5. Page eviction

In Lab 1, we did not correctly observe the limit on the maximum number of pages in the buffer pool defined by the constructor argument `numPages`. Now, you will choose a page eviction policy and instrument any previous code that reads or creates pages to implement your policy.

When more than `numPages` 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. Describe your policy in the lab writeup.

Notice that `BufferPool` asks you to implement a `flushAllPages()` 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 should never call this method from any real code. Because of the way we have implemented `ScanTest.cacheTest`, you will need to ensure that your `flushPage` and `flushAllPages` methods do not evict pages from the buffer pool to properly pass this test. `flushAllPages` should call `flushPage` on all pages in the `BufferPool`, and `flushPage` should write any dirty page to disk and mark it as not dirty, while leaving it in the `BufferPool`. The only method which should remove page from the buffer pool is `evictPage`, which should call `flushPage` on any dirty page it evicts.

Exercise 5. Fill in the `flushPage()` method and additional helper methods to implement page eviction in:

- `src/simplydb/BufferPool.java`

If you did not implement `writePage()` in `HeapFile.java` above, you will also need to do that here.

At this point, your code should pass the `EvictionTest` system test.

Since we will not be checking for any particular eviction policy, this test works by creating a `BufferPool` with 16 pages (NOTE: while `DEFAULT_PAGES` is 50, we are initializing the `BufferPool` with less!), scanning a file with many more than 16 pages, and seeing if the memory usage of the JVM increases by more than 5 MB. If you do not implement an eviction policy correctly, you will not evict enough pages, and will go over the size limitation, thus failing the test.

You have now completed this lab. Good work!

2.6. Query walkthrough

The following code implements a simple join query between two tables, each consisting of three columns of integers. (The file `some_data_file1.dat` and `some_data_file2.dat` are binary representation of the pages from this file). This code is equivalent to the SQL statement:

```
SELECT *
FROM some_data_file1, some_data_file2
WHERE some_data_file1.field1 = some_data_file2.field1
AND some_data_file1.id > 1
```

For more extensive examples of query operations, you may find it helpful to browse the unit tests for joins, filters, and aggregates.

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

public class jointest {

    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 td = new TupleDesc(types, names);

        // create the tables, associate them with the data files
        // and tell the catalog about the schema the tables.
        HeapFile table1 = new HeapFile(new File("some_data_file1.dat"), td);
        Database.getCatalog().addTable(table1, "t1");

        HeapFile table2 = new HeapFile(new File("some_data_file2.dat"), td);
        Database.getCatalog().addTable(table2, "t2");

        // construct the query: we use two SeqScans, which spoonfeed
```

```

// tuples via iterators into join
TransactionId tid = new TransactionId();

SeqScan ssl = new SeqScan(tid, table1.getId(), "t1");
SeqScan ss2 = new SeqScan(tid, table2.getId(), "t2");

// create a filter for the where condition
Filter sfl = new Filter(
    new Predicate(0,
        Predicate.Op.GREATER_THAN, new IntField(1)), ss1);

JoinPredicate p = new JoinPredicate(1, Predicate.Op.EQUALS, 1);
Join j = new Join(p, sfl, ss2);

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

```

Both tables have three integer fields. To express this, we create a `TupleDesc` object and pass it an array of `Type` objects indicating field types and `String` objects indicating field names. Once we have created this `TupleDesc`, we initialize two `HeapFile` objects representing the tables. Once we have created the tables, we add them 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 `SeqScan` operators that scan the tuples from each file on disk, connected to a `Filter` operator on the first `HeapFile`, connected to a `Join` operator that joins the tuples in the tables according to the `JoinPredicate`. 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., `Join`). The test program then repeatedly calls `next` on the `Join` operator, which in turn pulls tuples from its children. As tuples are output from the `Join`, they are printed out on the command line.

2.7. Query Parser and Contest

We've provided you with a query parser for SimpleDB that you can use to write and run SQL queries against your database once you have completed the exercises in this lab.

The first step is to create some data tables and a catalog. Suppose you have a file `data.txt` with the following contents:

```

1,10
2,20
3,30
4,40
5,50
5,50

```

You can convert this into a SimpleDB table using the `convert` command (make sure to type `ant` first!):

```
java -jar dist/simpledb.jar convert data.txt 2 "int,int"
```

This creates a file `data.dat`. In addition to the table's raw data, the two additional parameters specify that each record has two fields and that their types are `int` and `int`.

Next, create a catalog file, `catalog.txt`, with the follow contents:

```
data (f1 int, f2 int)
```

This tells SimpleDB that there is one table, `data` (stored in `data.dat`) with two integer fields named `f1` and `f2`.

Finally, invoke the parser. You must run `java` from the command line (`ant` doesn't work properly with interactive targets.) From the `simpledb/` directory, type:

```
java -jar dist/simpledb.jar parser catalog.txt
```

You should see output like:

```
Added table : data with schema INT(f1), INT(f2),
SimpleDB>
```

Finally, you can run a query:

```
SimpleDB> select d.f1, d.f2 from data d;
Started a new transaction tid = 1221852405823
  ADDING TABLE d(data) TO tableMap
      TABLE HAS tupleDesc INT(d.f1), INT(d.f2),
1         10
2         20
3         30
4         40
5         50
5         50

 6 rows.
-----
0.16 seconds

SimpleDB>
```

The parser is relatively full featured (including support for `SELECT`s, `INSERT`s, `DELETE`s, and transactions), but does have some problems and does not necessarily report completely informative error messages. Here are some limitations to bear in mind:

- You must preface every field name with its table name, even if the field name is unique (you can use table

- name aliases, as in the example above, but you cannot use the AS keyword.)
- Nested queries are supported in the WHERE clause, but not the FROM clause.
- No arithmetic expressions are supported (for example, you can't take the sum of two fields.)
- At most one GROUP BY and one aggregate column are allowed.
- Set-oriented operators like IN, UNION, and EXCEPT are not allowed.
- Only AND expressions in the WHERE clause are allowed.
- UPDATE expressions are not supported.
- The string operator LIKE is allowed, but must be written out fully (that is, the Postgres tilde [~] shorthand is not allowed.)

Contest (Optional)

The first-place winner in the contest will receive a \$50 gift certificate to Amazon. The second-place winner will receive a \$25 gift certificate to Amazon.

We have built SimpleDB-encoded version of the NSF database you used in problem set 1; the needed files are located at: http://db.csail.mit.edu/nsf_data.tar.gz

You should download the file and unpack it. It will create four files in the `nsf_data` directory. Move them into the `simpledb` directory. The following commands will accomplish this, if you run them from the `simpledb` directory:

```
wget http://db.csail.mit.edu/nsf_data.tar.gz
tar xvzf nsf_data.tar.gz
mv nsf_data/* .
rm -r nsf_data.tar.gz nsf_data
```

You can then run the parser with:

```
java -jar dist/simpledb.jar parser nsf.schema
```

We will give a prize to the submission that has shortest total runtime for the following three queries (where total runtime is the sum of the runtime of each of the individual queries):

```
1. SELECT g.title
   FROM grants g
   WHERE g.title LIKE 'Monkey';
```

```
2. SELECT g.title
   FROM grants g, researchers r, grant_researchers gr
   WHERE r.name = 'Samuel Madden'
   AND gr.researcherid = r.id
   AND gr.grantid = g.id;
```

```

3. SELECT r2.name, count(g.id)
   FROM grants g, researchers r, researchers r2, grant_researchers gr,
   grant_researchers gr2
   WHERE r.name = 'Samuel Madden'
   AND gr.researcherid = r.id
   AND gr.grantid = g.id
   AND gr2.researcherid = r2.id
   AND gr.grantid = gr2.grantid
   GROUP BY r2.name
   ORDER BY r2.name;

```

Note that each query will print out its runtime after it executes.

You may wish to create optimized implementations of some of the operators; in particular, a fast join operator (e.g., not nested loops) will be essential for good performance on queries 2 and 3.

There is currently no optimizer in the parser, so the queries above have been written to cause the parser generate reasonable plans. Here are some helpful hints about how the parser works that you may wish to exploit while running these queries:

- The table on the left side of the joins in these queries is passed in as the first `DbIterator` parameter to `Join`.
- Expressions in the `WHERE` clause are added to the plan from top to bottom, such that first expression will be the bottom-most operator in the generated query plan. For example, the generated plan for Query 2 is:

```
Project(Join(Join(Filter(a),pa),p))
```

Our reference implementation can run Query 1 in about .1 seconds, Query 2 in about .4 seconds, and Query 3 in about .7 seconds. We implemented a special-purpose join operator for equality joins but did little else to optimize performance.

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. If you used something other than a nested-loops join, describe the tradeoffs of the algorithm you chose.
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the lab, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

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

3.2. Submitting your assignment

To submit your code, please create a `6.830-lab2.tar.gz` tarball (such that, untarred, it creates a `6.830-lab2/src/simplydb` directory with your code) and submit it. You may submit your code multiple times; we will use the latest version you submit that arrives before the deadline (before 11:59pm on the due date). If applicable, please indicate your partner in your writeup. Please also submit your individual writeup as a PDF or plain text file (.txt). Please do not submit a .doc or .docx. Make sure your code is packaged so the instructions outlined in section 3.4 work.

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 `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

50% 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 produces no errors (passes all of the tests) from both `ant test` and `ant systemtest`.

Important: before testing, we will replace your `build.xml`, `HeapFileEncoder.java`, 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 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-lab2.tar.gz
$ tar xvf 6.830-lab2.tar
$ cd ./6.830-lab2
[replace build.xml, HeapFileEncoder.java, 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 50% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

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

MIT OpenCourseWare
<http://ocw.mit.edu>

6.830 / 6.814 Database Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.