Search this site

Intro to Database Systems - Fall 2013
Announcements
Bunnies
Class Schedule and Lecture Notes
Course Information
Some Terminology
Homeworks
Project 1
Project 2
Project 3
Project 4
Recovery and Database Design Homework
SQL Homework
Section Notes
Sitemap

Homeworks >

Project 3

CS186 Project 3: Query Optimization

CS186, UC Berkeley, Fall 2013
Points: 10% of your final grade
Note: This project is to be done alone or in pairs!
Assigned: Tuesday, October 22, 2013
Due: Thursday, November 7, 2013 11:59PM (You have 4 slip days for all CS186 projects, use them wisely)

- 10/22/13 : Initial version
- 10/27/13 : Update build.xml for correcting "ant handin". Change some notations in Section 2.2.1.

Project Description

In this project, you will implement a query optimizer on top of SimpleDB. The main tasks include implementing a selectivity estimation framework and a cost-based optimizer. You have freedom as to exactly what you implement, but we recommend using something similar to the Selinger cost-based optimizer discussed in class. The remainder of this document describes what is involved in adding optimizer support and provides a basic outline of how you might add this support to your database.
As with the previous projects, we recommend that you start as early as possible.

1. Getting started

You should begin with the code you submitted for Project 2. (If you did not submit code for Project 2, or your solution didn't work properly, contact us to discuss options.) We have provided you with extra test cases as well as source code files for this project 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 copy of your Project 2 solution by typing :

  $ cp -r CS186-proj2 CS186-proj3

- Download the new tests and skeleton code for Project 3 (See the bottom of the page for downloads)

- Extract the new files for Project 2 by typing:

  tar -xvzf CS186-proj3-supplement.tar.gz

We highly recommend that you develop on the same Virtual Machine you used for previous projects.

1.1. Implementation hints

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 the tests you will need to pass.

Here's a rough outline of one way you might proceed with this project. More details on these steps are given in Section 2 below.

- Implement the methods in the `TableStats` class that allow it to estimate selectivities of filters and cost of scans, using histograms (skeleton provided for the `IntHistogram` class) or some other form of statistics of your devising.
- Implement the methods in the `JoinOptimizer` class that allow it to estimate the cost and selectivities of joins.
- Write the `orderJoins` method in `JoinOptimizer`. This method must produce an optimal ordering for a series of joins (likely using dynamic programming), given statistics computed in the previous two steps.

2. Optimizer outline

Recall that the main idea of a cost-based optimizer is to:

- Use statistics about tables to estimate "costs" of different query plans. Typically, the cost of a plan is related to the cardinalities of (number of tuples produced by) intermediate joins and selections, as well as the selectivity of filter and join predicates.
- Use these statistics to order joins and selections in an optimal way, and to select the best implementation for join algorithms from amongst several alternatives.

In this project, you will implement code to perform both of these functions.

The optimizer will be invoked from `simpledb/Parser.java`. When the Parser is invoked, it will compute statistics over all of the tables (using statistics code you provide). When a query is issued, the parser will convert the query into a logical plan representation and then call your query optimizer to generate an optimal plan.

2.1 Overall Optimizer Structure

Before getting started with the implementation, you need to understand the overall structure of the SimpleDB optimizer.

The overall control flow of the SimpleDB modules of the parser and optimizer is shown in Figure 1.

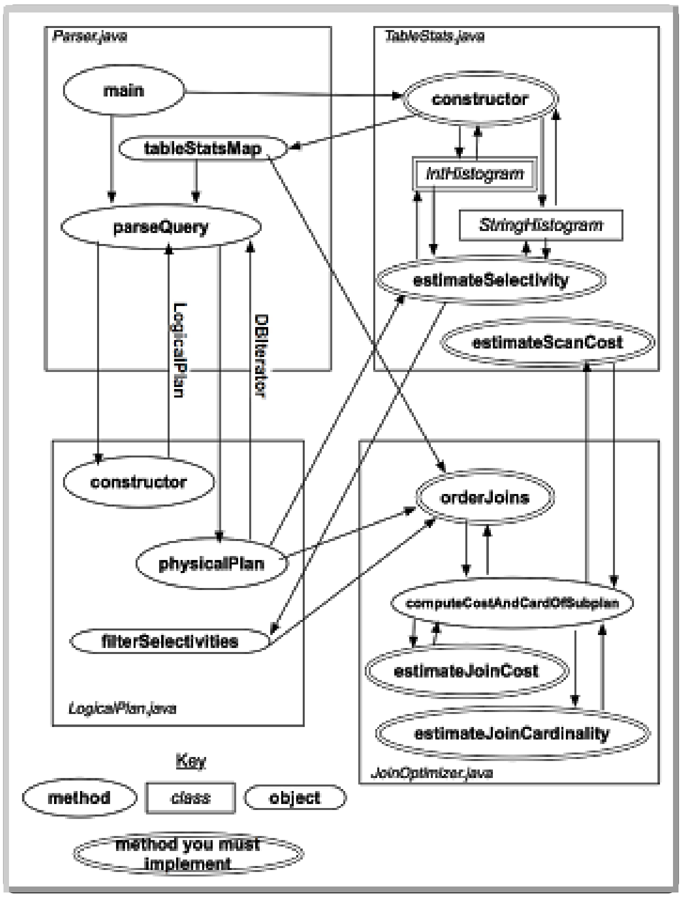

Figure 1: Diagram illustrating classes, methods, and objects used in the parser and optimizer.

The key at the bottom explains the symbols; you will implement the components with double-borders. The classes and methods will be explained in more detail in the text that follows (you may wish to refer back to this diagram), but the basic operation is as follows:

1. `Parser.java` constructs a set of table statistics (stored in the `statsMap` container) when it is initialized. It then waits for a query to be input, and calls the method `parseQuery` on that query.
2. `parseQuery` first constructs a `LogicalPlan` that represents the parsed query. `parseQuery` then calls the method `physicalPlan` on the `LogicalPlan` instance it has constructed. The `physicalPlan` method returns a `DBIterator` object that can be used to actually run the query.

In the exercises to come, you will implement the methods that help `physicalPlan` devise an optimal plan.

Exercise 1: Parser.java

When you launch SimpleDB, the entry point of the application is simpledb.Parser.main().

Starting from that entry point, describe the life of a query from its submission by the user to its execution. For this, list the sequence of methods that are invoked. For each method, describe its primary functions. When you describe the methods that build the physical query plan, **discuss how the plan is built**.

To help you, we provide the description of the first few methods. In general, however, it is up to you to decide on the appropriate level of detail for your description. Keep in mind that the goal is to demonstrate your understanding of the *optimizer*.

Life of a query in SimpleDB

Step 1: simpledb.Parser.main() and simpledb.Parser.start()

simpledb.Parser.main() is the entry point for the SimpleDB system. It calls simpledb.Parser.start(). The latter performs three main actions:

- It populates the SimpleDB catalog from the catalog text file provided by the user as argument (Database.getCatalog().loadSchema(argv[0]);).
- For each table defined in the system catalog, it computes statistics over the data in the table by calling: TableStats.computeStatistics(), which then does: TableStats s = new TableStats(tableid, IOCOSTPERPAGE);
- It processes the statements submitted by the user (processNextStatement(new ByteArrayInputStream(statementBytes));)

Step 2: simpledb.Parser.processNextStatement()

This method takes two key actions:

- First, it gets a physical plan for the query by invoking handleQueryStatement((ZQuery)s);
- Then it executes the query by calling query.execute();

Step 3: simpledb.Parser.handleQueryStatement()

In your writeup, please continue describing the life of the query in the SimpleDB system. Remember to describe the key steps involved in the construction of the physical query plan.

2.2. Statistics Estimation

Accurately estimating plan cost is quite tricky. In this project, we will focus only on the cost of sequences of joins and base table accesses. We won't worry about access method selection (since we only have one access method, table scans) or the costs of additional operators (like aggregates). You are only required to consider left-deep plans for this project although we provide methods that will help you search through a larger variety of plans, which is the set of all *linear* plans. With linear plans, the relation on one side of each operator is always a base relation but it can appear either as the outer or inner relation.

2.2.1 Overall Plan Cost

We will write join plans of the form `p=t1 join t2 join ... tn` , which signifies a left deep join where t1 is the left-most join (deepest in the tree). Given a plan like `p`, its cost can be expressed as:

```
iocost(t1) + iocost(t2) + cpucost(t1 join t2) +
iocost(t3) + cpucost((t1 join t2) join t3) +
```

```
...
```

Here, `iocost(t1)` is the I/O cost of scanning table t1, `cpucost(t1 join t2)` is the CPU cost to join t1 to t2. To make I/O and CPU cost comparable, typically a constant scaling factor is used, e.g.:

```
cost(predicate application) = 1
cost(pageScan) = SCALING_FACTOR x cost(predicate application)
```

For this project, you can ignore the effects of caching (e.g., assume that every access to a table incurs the full cost of a scan). Therefore, `iocost(t1)` is simply `number_IO_for_scanning_t1 x SCALING_FACTOR`.

2.2.2 Join Cost

When using nested loops joins, recall that the cost of a join between two tables t1 and t2 (where t1 is the outer) is simply:

```
joincost(t1 join t2) = scancost(t1) + ntups(t1) x scancost(t2)
                       + ntups(t1) x ntups(t2)  //CPU cost
```

Here, `ntups(t1)` is the number of tuples in table t1.

2.2.3 Filter Selectivity

The value of `ntups` can be directly computed for a base table by scanning that table. Estimating `ntups` for a table with one or more selection predicates over it can be trickier -- this is the *filter selectivity estimation* problem. Here's one approach that you might use, based on computing a histogram over the values in the table:

- Compute the minimum and maximum values for every attribute in the table (by scanning it once).
- Construct a histogram for every attribute in the table. A simple approach is to use a fixed number of buckets *NumB*, with each bucket representing the number of records in a fixed range of the domain of the attribute of the histogram. For example, if a field *f* ranges from 1 to 100, and there are 10 buckets, then bucket 1 might contain the count of the number of records between 1 and 10, bucket 2 a count of the number of records between 11 and 20, and so on.
- Scan the table again, selecting out all of fields of all of the tuples and using them to populate the counts of the buckets in each histogram.
- To estimate the selectivity of an equality expression, *f=const*, compute the bucket that contains value *const*. Suppose the width (range of values) of the bucket is *w*, the height (number of tuples) is *h*, and the number of tuples in the table is *ntups*. Then, assuming values are uniformly distributed throughout the bucket, the selectivity of the expression is roughly *(h / w) / ntups* , since *(h/w)* represents the expected number of tuples in the bin with value *const*.
- To estimate the selectivity of a range expression *f>const*, compute the bucket *b* that *const* is in, with width *w_b* and height *h_b*. Then, *b* contains a fraction *b_f = h_b / ntups* of the total tuples. Assuming tuples are uniformly distributed throughout *b*, the fraction *b_part* of *b* that is > *const* is *(b_right - const) / w_b*, where *b_right* is the right endpoint of *b*'s bucket. Thus, bucket *b* contributes *(b_f x b_part)* selectivity

to the predicate. In addition, buckets *b+1...NumB-1* contribute all of their selectivity (which can be computed using a formula similar to *b_f* above). Summing the selectivity contributions of all the buckets will yield the overall selectivity of the expression. Figure 2 illustrates this process.

- Selectivity of expressions involving *less than* can be performed similar to the greater than case, looking at buckets down to 0.

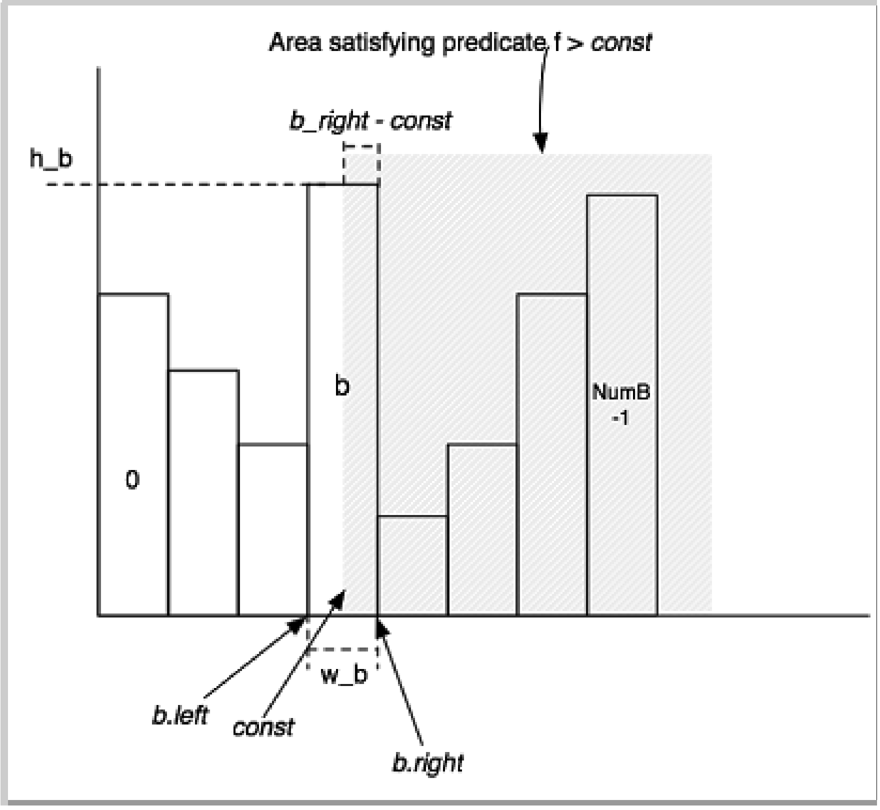

Figure 2: Diagram illustrating the histograms you will implement in this project.

In the next two exercises, you will code to perform selectivity estimation of joins and filters.

Exercise 2: IntHistogram.java

You will need to implement some way to record table statistics for selectivity estimation. We have provided a skeleton class, `IntHistogram` that will do this. Our intent is that you calculate histograms using the bucket-based method described above, but you are free to use some other method so long as it provides reasonable selectivity estimates.

We have provided a class `StringHistogram` that uses `IntHistogram` to compute selectivites for String predicates. You may modify `StringHistogram` if you want to implement a better estimator, although it's not necessary for completing this project.

After completing this exercise, you should be able to pass the `IntHistogramTest` unit test.

Exercise 3: TableStats.java

The class `TableStats` contains methods that compute the number of tuples and pages in a table and that estimate the selectivity of predicates over the fields of that table. The query parser we have created creates one instance of `TableStats` per table, and passes these structures into your query optimizer (which you will need in later exercises).

You should fill in the following methods and classes in `TableStats`:

- Implement the `TableStats` constructor: Once you have implemented a method for tracking statistics such as histograms, you should implement the `TableStats` constructor, adding code to scan the table (possibly multiple times) to build the statistics you need.
- Implement `estimateSelectivity(int field, Predicate.Op op, Field constant)`: Using your statistics (e.g., an `IntHistogram` or `StringHistogram` depending on the type of the field), estimate the selectivity of predicate `field op constant` on the table.
- Implement `estimateScanCost()`: This method estimates the cost of sequentially scanning the file, given that the cost to read a page is `costPerPageIO`. You can assume that there are no seeks and that no pages are in the buffer pool. This method may use costs or sizes you computed in the constructor.
- Implement `estimateTableCardinality(double selectivityFactor)`: This method returns the number of tuples in the relation, given that a predicate with selectivity selectivityFactor is applied. This method may use costs or sizes you computed in the constructor.

You may wish to modify the constructor of `TableStats.java` to, for example, compute histograms over the fields as described above for purposes of selectivity estimation.

After completing these tasks you should be able to pass the unit tests in `TableStatsTest`.

2.2.4 Join Cardinality

Finally, observe that the cost for the join plan `p` above includes expressions of the form `joincost((t1 join t2) join t3)`. To evaluate this expression, you need some way to estimate the size (`ntups`) of `t1 join t2`. This *join cardinality estimation* problem is harder than the filter selectivity estimation problem. In this project, you aren't required to do anything fancy for this. While implementing your simple solution, you should keep in mind the following:

- For equality joins, when one of the attributes is a primary key, the number of tuples produced by the join cannot be larger than the cardinality of the non-primary key attribute.
- For equality joins when there is no primary key, it's hard to say much about what the size of the output is -- it could be the size of the product of the cardinalities of the tables (if both tables have the same value for all tuples) -- or it could be 0. It's fine to make up a simple heuristic (say, the size of the larger of the two tables).
- For range scans, it is similarly hard to say anything accurate about sizes. The size of the output should be proportional to the sizes of the inputs. It is fine to assume that a fixed fraction of the cross-product is emitted by range scans (say, 30%). In general, the cost of a range join should be larger than the cost of a non-primary key equality join of two tables

of the same size.

Exercise 4: Join Cost Estimation

The class `JoinOptimizer.java` includes all of the methods for ordering and computing costs of joins. In this exercise, you will write the methods for estimating the selectivity and cost of a join, specifically:

- Implement `estimateJoinCost(LogicalJoinNode j, int card1, int card2, double cost1, double cost2)`: This method estimates the cost of join j, given that the left input is of cardinality card1, the right input of cardinality card2, that the cost to access the left input is cost1, and that the cost to access the right input is cost2. You can assume the join is an NL join, and apply the formula mentioned earlier.
- Implement `estimateJoinCardinality(LogicalJoinNode j, int card1, int card2, boolean t1pkey, boolean t2pkey)`: This method estimates the number of tuples output by join j, given that the left input is size card1, the right input is size card2, and the flags t1pkey and t2pkey that indicate whether the left and right (respectively) field is unique (a primary key).

After implementing these methods, you should be able to pass the unit tests in `JoinOptimizerTest.java`, other than `orderJoinsTest`.

2.3 Join Ordering

Now that you have implemented methods for estimating costs, you will implement a Selinger-style optimizer. For these methods, joins are expressed as a list of join nodes (e.g., predicates over two tables) as opposed to a list of relations to join as described in class.

Translating the algorithm to the join node list form mentioned above, an outline in pseudocode would be as follows.

Hint: We discussed this algorithm in detail in class!

```
j = set of join nodes
for (i in 1...|j|):  // First find best plan for single join, ther
    for s in {all length i subsets of j} // Looking at a concrete
      bestPlan = {}  // We want to find the best plan for this cor
      for s' in {all length i-1 subsets of s}
           subplan = optjoin(s')  // Look-up in the cache the best
           plan = best way to join (s-s') to subplan // Now find t
           if (cost(plan) < cost(bestPlan))
              bestPlan = plan // Update the best plan for computir
      optjoin(s) = bestPlan
return optjoin(j)
```

To help you implement this algorithm, we have provided several classes and methods to assist you. First, the method `enumerateSubsets(Vector v, int size)` in `JoinOptimizer.java` will return a set of all of the subsets of `v` of size `size`. This method is not particularly efficient; you can try to implement a more efficient enumerator by yourself, but it's not necessary for this project.

Second, we have provided the method:

```
    private CostCard computeCostAndCardOfSubplan(HashMap<Strir
                                                 HashMap<String
```

```
                                    LogicalJoinNoc
                                    Set<LogicalJoi
                                    double bestCos
                                    PlanCache pc)
```

Given a subset of joins (`joinSet`), and a join to remove from this set (`joinToRemove`), this method computes the best way to join `joinToRemove` to `joinSet - {joinToRemove}`. It returns this best method in a `CostCard` object, which includes the cost, cardinality, and best join ordering (as a vector). `computeCostAndCardOfSubplan` may return null, if no plan can be found (because, for example, there is no linear join that is possible), or if the cost of all plans is greater than the `bestCostSoFar` argument. The method uses a cache of previous joins called `pc` (`optjoin` in the psuedocode above) to quickly lookup the fastest way to join `joinSet - {joinToRemove}`. The other arguments (`stats` and `filterSelectivities`) are passed into the `orderJoins` method that you must implement as a part of Exercise 4, and are explained below. This method essentially performs lines 6--8 of the psuedocode described earlier.

Note: While the original Selinger optimizer considered only left-deep plans, computeCostAndCardOfSubplan considers all *linear* plans.

Third, we have provided the method:

```
private void printJoins(Vector<LogicalJoinNode> js,
                        PlanCache pc,
                        HashMap<String, TableStats> stats,
                        HashMap<String, Double> selectiviti
```

This method can be used to display a graphical representation of the join costs/cardinalities (when the "explain" flag is set via the "-explain" option to the optimizer, for example).

Fourth, we have provided a class `PlanCache` that can be used to cache the best way to join a subset of the joins considered so far in your implementation of the Selinger-style optimizer (an instance of this class is needed to use `computeCostAndCardOfSubplan`).

Exercise 5: Join Ordering

In `JoinOptimizer.java`, implement the method:

```
Vector orderJoins(HashMap<String, TableStats> stats,
                  HashMap<String, Double> filterSelectivities
                  boolean explain)
```

This method should operate on the `joins` class member, returning a new Vector that specifies the order in which joins should be done. Item 0 of this vector indicates the bottom-most join in a linear plan. Adjacent joins in the returned vector should share at least one field to ensure the plan is linear. Here `stats` is an object that lets you find the `TableStats` for a given table name that appears in the `FROM` list of the query. `filterSelectivities` allows you to find the selectivity of any predicates over a table; it is guaranteed to have one entry per table name in the `FROM` list. Finally, `explain` specifies that you should output a representation of the join order for informational purposes.

You may wish to use the helper methods and classes described above to assist in your implementation. Roughly, your implementation should follow the pseudocode above, looping through subset sizes, subsets, and sub-plans of subsets, calling `computeCostAndCardOfSubplan` and building a `PlanCache` object that stores the minimal-cost way to perform each subset join.

After implementing this method, you should be able to pass the test `OrderJoinsTest`. You should also pass the system test `QueryTest`.

2.4 Putting it all together

Now that you have a working optimizer, you can study the query plans that your optimizer generates. In this exercise, we will use the IMDB dataset. The database consists of six tables:

```
Actor (id, fname, lname, gender)
Movie (id, name, year)
Director (id, fname, lname)
Casts (pid, mid, role) // Indicates which actor (pid references Actor.id
Movie_Director (did, mid) // Indicates which director (did references Di
Genre (mid, genre)
```

We provide you with the following (see the bottom of the page for downloads):

- *imdb.schema*: the schema of the IMDB database:
- sample-0.01.tar.bz2: a small 1% version of the IMDB database. This is the recommended version to use for this project.
- sample-0.001.tar.bz2: we also provide 0.1% sample, in case your version of SimpleDB is too slow to handle the 1% dataset.
- sample-0.1.tar.bz2: the 10% sample, if you are adventurous.

The QueryPlanVisualizer will print the whole query plan for each query. If you would like to see more information about the joins, you can launch SimpleDB with the -explain option enabled:

```
java -classpath "bin/src/:lib/*" simpledb.Parser $IMDB_DATA_FC
```

Excercise 6: Query Plans

6.1 Execute the following query

```
select d.fname, d.lname
from Actor a, Casts c, Movie_Director m, Director d
where a.id=c.pid and c.mid=m.mid and m.did=d.id
and a.fname='John' and a.lname='Spicer';
```

Show the query plan that your optimizer selected. Explain why your optimizer selected that plan. Be careful as the plan may be different for the 1%, 0.1%, and 10% datasets (you do not need to test with all the datasets, just pick one).

6.2 Execute another SQL query of your choice over the IMDB database. Show the query plan that your optimizer generates. Discuss why your optimizer generates that plan. Try to find an interesting SQL query with a combination of joins and selections.

You have now completed this project. Good work!

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- **Answer the questions in Exercise 1 and Exercise 6.**
- 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 project, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

All CS186 projects are to be completed individually or with a partner.

3.2. Submitting your assignment

The files you need to hand in are:

1. `CS186-proj3.tar.gz` tarball (such that, untarred, it creates a `CS186-proj3/src/java/simpledb` directory with your code). You can use the `ant handin` target to generate the tarball.
2. `answers.txt` containing the writeup.
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 by glookup when you run the submit proj3.

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). **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 proj3`:

```
% submit proj3
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-proj3.tar.gz.
Submitting README.
The files you have submitted are:
./MY.PARTNERS ./README ./CS186-proj3.tar.gz ./answers.txt
Is this correct? [yes/no] y
Copying submission of assignment proj3....
Submission complete.
```

Make sure your code is packaged so the instructions outlined in sectio

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! You can also post on Piazza if you feel you have run into a bug.

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`, `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 CS186-proj3.tar.gz
$ tar xvf CS186-proj3.tar
$ cd ./CS186-proj3
[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 25% 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!

Acknowledgements

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

CS186-proj3-s…	Liwen Sun,	Oct 2	v.1	
imdb.schema (0k	Liwen Sun,	Oct 2	v.2	
sample-0.001.t…	Liwen Sun,	Oct 2	v.2	

sample-0.01.ta…	Liwen Sun, Oct 2	v.2	
sample-0.1.tar…	Liwen Sun, Oct 2	v.2	

Comments