In this assignment, you will generate data and compare two systems to see how they work for querying. To use the Courant systems, take a look at: http://cims.nyu.edu/webapps/content/systems/userservices/

Here are some instructions for self-installing (from Rafi Shamim):

- Install homebrew from http://brew.sh/
- Then the following commands:
  $ brew update
  $ brew upgrade
  $ brew install mysql

To start the database, use the mysql.server command.
The mysql command allows you to connect.

1. For the tests on each system you will receive 20 points. If you manage to test both and give a comparison, you will receive 20 + 20 + 10 (for the comparison). We will run your code on our data as well as look at the results on your data. Our data will have the same schema and the same random distributions.

Design a program in your favorite programming language that can generate values based on a fractal probability distribution (70-30 rule). An example function is pseudocode to do this is as follows:

```
gen(frac, N)
begin
  p:= random permutation of numbers from 1 to N
  outvec:= p // so outvec is of length N
  while(|p| > 1
    p:= first frac*|p| elements of p // round down
    concatenate p to outvec
  end while
  return random permutation of outvec
end gen
```

Here is an example:

|p| is just the size of vector p.
so let's say that p is 7 4 6 2 3 8 1 5
|p| = 8
Initially outvec is the same as p
if frac is 0.5 then the first loop will
make p = 7 4 6 2
and outvec = 7 4 6 2 3 8 1 5 7 4 6 2
After the second iteration,
p = 7 4  
and outvec
7 4 6 2 3 8 1 5 7 4 6 2 7 4
Finally after the last iteration
outvec = 7 4 6 2 3 8 1 5 7 4 6 2 7 4 7

Execute this program with gen(0.3, x) where x is 70,000 or so (might be more). The idea is to generate an array of 100,000 fractally distributed stock symbols.

Populate table trade(stocksymbol, time, quantity, price) such that there are 100,000 stock symbols, quantities uniformly ranging from 100 to 10,000, and prices uniformly ranging in some five point interval between 50 and 500, so successive prices should vary by at least 1 but no more than 5. Time is just a numerical value. The trade table should have 10 million rows with time as the only key.

Next write and time the following queries (send in your typescript file that you get from the script command) in your favorite sql dialect (but it would be easiest in AQuery). Note that you can answer each question with a series of queries that involve temp tables if you wish.

2. (30 points) Choose two rules of thumb from the book (e.g., rules having to do with chopping, indexes, commits, checkpoints, etc.). For each one, find (and demonstrate using e.g. mysql and kdb). data distributions/access patterns where it is satisfied and data distributions/access patterns where it is not. A data distribution refers to the frequency of occurrences of different values of a given variable across the rows - e.g. uniform and fractal are two different (and convenient since you already generated data with them) distributions. Thus, you need to show four different performance numbers for each rule of thumb derived by executing real queries in a real database: results with and without the rule of thumb applied to columns with two different data distributions. Comment on whether the case (i.e. the data distribution) where the rule of thumb is satisfied is more likely than when it isn’t. Also explore how you would modify the statement of the rule of thumb to make it more precise given the performance results.

3. (20 points) This is a slight adaptation of a challenge posed by Peter Boncz. In a social network context, you are given

- a table relating individuals to performing artists they like. like(person, artist).
- A table identifying the friends of each user friend(person1, person2).

These tables are all very large (we will generate them for you). The challenge is to use mysql, sqlite or AQuery to find the most efficient way to solve the following problem: for each user u and each artist a that the user does not yet like, find friends of u who like a. So the output
would consist of triplets of the form \((u_1, u_2, a)\), where \(u_1\) has a friend \(u_2\) who likes \(a\), but \(u_1\) doesn’t yet like \(a\). You may use several queries along with temp tables if you wish. \(u_1\) likes \(a_1\) and \(a_2\) and \(a_3\), \(u_1\) and \(u_2\) are friends, \(u_2\) likes \(a_3\) and \(a_4\). Then we should get \((u_1, u_2, a_4), (u_2, u_1, a_1), (u_2, u_1, a_2)\).