Advanced Database Systems - CSCI-GA.2434 - Fall 2021

Professor: Dennis Shasha

Homework 2 - Due: at 4:30 PM Eastern Standard Time, Wednesday, November 24, 2021

Please submit as follows:

```
- Sign in to https://brightspace.nyu.edu with your NYU NetID and password.
- Under the 'My Courses' section, select the course 'Advanced Database Systems CSCI-GA.2434-00
- Navigate to the 'Assignment' tab.
- Please submit your solution as a .pdf file/.zip file.
```

In this assignment, you will generate data and compare two systems to see how they work for querying.

All assignments go to the classes website. Here are some instructions for self-installing if you want mySQL to be local.

```
- 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.
```

Here are some notes on mysql http://cs.nyu.edu/courses/fall17/CSCI-GA.2434-001/mysqlnotes

To get a mysql database going on the NYU servers, please follow the instructions here: https://cims.nyu.edu/webapps/content/systems/userservices/databases-selfmanaged

The teaching assistant will check this out and report any problems to cims support.

For this assignment, please use the number-crunching machines for this type of work. More information about available compute servers is listed here: https://cims.nyu.edu/webapps/content/systems/resources/computeservers

For postgres, either download the latest version to your local machine. To load on the cims servers, follow this link. http://cs.nyu.edu/courses/fall21/CSCI-GA.2434-001/postgresonlinserv.pdf

1. For the tests on the system of your choice, you will receive 20 points. There is no extra credit at all. You can use any system you like. 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
      prepend p to outvec
end while
return random permutation of outvec
```

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 prepending to outvec = 7 4 6 2 7 4 6 2 3 8 1 5
After the second iteration,
p = 7 4
and prepend to outvec
7 4 7 4 6 2 7 4 6 2 3 8 1 5
Finally after the last iteration
outvec = 7 7 4 7 4 6 2 7 4 6 2 3 8 1 5
```

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) from those 100,000 stock symbols (some of which will be duplicates) and successive trades will apply to one symbol chosen randomly and uniformly with replacement from those 100,000 symbols. Quantities should be uniformly ranging from 100 to 10,000, and prices uniformly ranging in some between 50 and 500. Successive prices for the same symbol should vary by at least 1 but no more than 5 and should still stay within that interval (so should never go to 49 or less or to 501 or more). Trades for different symbols should be interleaved.

Time is just a numerical value. No two trades should have the same numerical value. The trade table should have 10 million rows with time as the only key. Here is a tiny example (note that the successive prices of s1 never differ by more than 5 and similarly for s2)

```
s1, 1, 5000, 70
s2, 2, 4000, 100
s1, 3, 9000, 72
s3, 4, 3000, 200
s1, 5, 2000, 77
s2, 6, 9500, 96
s2, 7, 4500, 96
s1, 8, 3500, 79
s1, 9, 2500, 81
s1, 10, 5500, 76
```

Next write and time the following queries (send in your typescript file that you get from the UNIX script command) in your favorite sql dialect (but it would be easiest in AQuery) or other language (like pandas). Note that you can answer each question with a series of queries that involve temp tables if you wish. For full credit, each of these queries should take under two minutes on both your datasets and ours.

(a) Find the weighted (by quantity) average price of each stock over the entire time series. Example: for stock s1 above,

```
s1, 1, 5000, 70
s1, 3, 9000, 72
s1, 5, 2000, 77
s1, 8, 3500, 79
s1, 9, 2500, 81
s1, 10, 5500, 76
```

[(5000*70) + (9000*72) + ... + (5500 *76)]/(5000+9000+ ... + 5500)

(b) Find the vector of 10 trade unweighted price moving averages (i.e. moving average of price) per stock. In the example above, the three trade unweighted moving average for s1 would be the three trade unweighted moving average of the prices 70, 72, 77, 79, 81, 76 which would be 70, 71, 73, 76, 79, 78.67 The 70 is just the first entry, the 71 is the average of 70 and 72, the 73 is the average of 70, 72 and 77, the 76 is the average of 72 77 79, the 79 is the average of 77 79 81, and the 78.67 is the average of 79 81 and 76.

(c) Find the vector of 10 trade weighted moving averages per stock. Looking again at the example above, to compute the weighted moving average we would multiply the price by the volume. For example, if we wanted the three-way moving average of s1, that would be the 3 way moving average of 5000*70, 9000*72, 2000*77, 3500*79, 2500*81, 5500*76. Resulting in (5000*70)/5000, ((5000*70) + (9000*72))/(5000+9000), ((5000*70) + (9000*72) + (2000*77))/(5000+9000+2000),, ((9000*72) + (2000*77) + (3500*79))/(9000+2000+3500), etc

(d) Find the single best buy first/sell later trade you could have done on each stock (your query should work on our data as well as yours). That is, for each stock, find the maximum positive price difference between a later sell and an earlier buy.

2. (40 points with the possibility of 10 points extra credit) Choose two suggestions (sometimes called "rules of thumb") from the notes (e.g., rules having to do with chopping, indexes, table design, commits, checkpoints, etc.). For each one, find (and demonstrate using queries on two systems e.g. mysql and aquery). data distributions, access paterns, or table designs where the rule of thumb holds strongly (e.g. gives a big performance improvement) and data distributions, access patterns, or table designs where it is not satisfied (e.g. gives a performance drop or gives a much smaller performance improvement). Ideally, you'd use the same rules of thumb on the two systems, but if you can't, then use different ones, but two per system. A data distribution refers to the frequency of occurences 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, for each suggestion and each system, you need to show four different performance number pairs (two data distributions, with rule of thumb and without rule of thumb) and their associated queries. Altogether there

will be 16 numbers divided into four groups of four: suggestion 1 on system A, suggestion 2 on system A, suggestion 1 on system B, suggestion 2 on system B. Note that you are never directly comparing system A with system B.

Comment on whether the situation (i.e. the data distribution/access pattern/ table design) in which the rule of thumb is satisfied strongly is more likely than when it isn't. Also explore how you would refine the statement of the suggestion/rule of thumb to make it work regardless of the data distribution if possible. If you manage to test two rules of thumb on two systems, you will receive 20 + 20 points. If one of the two systems is aquery, you receive 10 points extra credit.

3. (20 points) This is a slight adaptation of a challenge posed by Peter Boncz. Use only one database system for this one. 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). Note that p1 and p2 are friends if and only if there is a row (p1, p2) or a row (p2, p1).

   These tables are all very large (we will generate them for you and you can see samples linked from the course website). The challenge is to use mysql, postgres, sqlite or AQuery to find the most efficient way (should take 1.5 minutes or less on a normal laptop) 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. If you add in index, then index creation time should count. So the output would consist of triplets of the form (u1, u2, a), where u1 has a friend u2 who likes a, but u1 doesn't yet like a. You may use several queries along with temp tables if you wish. u1 likes a1 and a2 and a3, u1 and u2 are friends, u2 likes a3 and a4. Then we should get (u1, u2, a4), (u2, u1, a1), (u2, u1, a2). Further if u1 doesn't like a1 but friends u2 and u3 both like a1, then get (u1, u2, a1) and (u1, u3, a1).

   Note that if you use aquery and need to do a join on all columns in a table, aquery will complain, so you might have to use a workaround: https://cs.nyu.edu/courses/fall20/CSCI-GA.2434-001/likesaquery.a