CS6650 Fall 2019 Assignment 2

Overview

You work for Upic - a global acquirer of ski resorts that is homogenezing skiing around the world. Upic ski resorts all use RFID lift ticket readers so that every time a skiier gets on a ski lift, the time of the ride and the skier ID are recorded.

In this course, through a series of assignments, we'll build a scalable distributed cloud-based system that can record all lift rides from all Upic resorts. This data can then be used as a basis for data analysis, answering such questions as:
* which lifts are most heavily used?
* which skiers ride the most lifts?
* How many lifts do skiers ride on average per day at resort X?

In Assignment 2, we'll build the server and database. This will give us the foundation to start thinking about more complex features in assignments 3 and 4.

Implement the Server

The server API is specified using Swagger. You should have implemented stubs for your APIs in your servlet in assignment 1.

Next we need to design a database schema and deploy this to your MySQL RDS instance. Think carefully about the design as you need to support a write-heavy workload (see lab 5).

You then need build the servlet business logic to implement this API. Each API should:

1. Accept the parameters for each operations as per the specification
2. Do basic parameter validation, and return a 4XX response code and error message if invalid values/formats supplied
3. If the request is valid, do the appropriate data processing to read/write from the database
4. Construct the correct response message and return a 200/201 response code

Test each servlet API with POSTMAN or an equivalent HTTP testing tools.

Make sure you can load the resulting .war file onto your EC2 free tier instance you have created and configured in lab 1 and call the APIs successfully.

Modify the Client

We need to make a simple modification to our client to change the workload profile it generates in phase 3.

Every time the client sends a POST in phase 3, it should immediately issue a corresponding GET request using the same URL parameter values.
This essentially increases the number of requests you send in phase 3. We'll use this new client in the next task.

Performance Testing

As in assignment 1, we want to test your new server/database with our load generating client. Test with {32, 64, 128, 256} client threads and report the ourputs for each.

You will probably find you get database deadlocks. You will need to find a way to work around these through schema changes or request retries. Your tests should successfully execute every request.

Load Balancing

The previous section has a bottleneck in the single server instance. So let's try and add capacity to our system and see what happens,

Set up AWS Elastic Load Balancing using either *Application* or *Network* load balancers. Enable load balancing with 4 free tier EC2 instances and see what effect this has on your performance.

Depending on your data model, you may find you free tier RDS server becomes a bottleneck. If so then allocate a more powerful RDS instance and see what effect it has. Just watch your costs.

Again test with {32, 64, 128, 256} clients and compare your results again the ones from the previous section with a single server.

Collect Runtime Statistics

Note the new */statistics* endpoint in v1.1 of the API. If you have ignored this so far, that's fine. Now we need to design and build it.

The basic aim is to provide an API that tells the caller the mean and maximin latencies for each endpoint. To do this you need to inject performance statistics code in your servlets and persist response times so they can be queried in aggregate.
There are two tricky issues to consider:
1. You want your statistics logging to be as low overhead as possible so that it does not slow down your servlet in processing client requests.
2. Your server will progressively capture more statistics as it processes more requests. This could quickly become a huge amount of data. So you want to think about what is actually useful here to store.

There are many ways to address this problem. Make some sound assumptions and design your solutions around these. For example you may want to only calculate response time values using the last N requests so your data collection is bounded. You will almost certainly want cached values for mean and maximum. Cached where is a design issue ;)

Submission Requirements

Submit your work to Blackboard Assignment 2 as a pdf document. The document should contain:

1. The URL for your git repo.
2. A 1–2 page description of your server design. Include major classes, packages, relationships, whatever you need to convey concisely how your client works
3. Single Server Tests - run your client with 32, 64, 128 and 256 threads, with numSkiers=20000, numLifts=40 and numRuns=20. Include the output window of each run in your submission (showing the wall time and performance stats) and plot a simple chart showing the throughput and mean response by the number of threads
4. Load Balanced Server Tests - run the client as above, showing the output window for each run. Also generate a plot of throughput and mean response time against number of threads.
5. Runtime Statistics Collection: Run a single client test with 256 client threads and wait for it to display its outputs. In the command line window, immediately issue a cURL/wget command on the /skiers POST/GET endpoints your have been testing. Hand in the command line window showing the client and cURL/wget outputs.

Grading:

1. Server implementation working (10 points)
2. Server design description (5 points) - clarity of description, good design practies used
3. Single Server Tests - (10 points) - 1 point per run output, 1 point for the chart, 5 points for sensible results.
4. Load Balanced Tests - (10 points) - 1 point per run, 1 point for the chart. 5 points for sensible results.

5. Runtime statistics - (10 points) - Successful implementation (5 points). Values of statistics from cURL/wget afetr test are sensible.

Deadline: 10/29, 3pm PST

Back to Course Home Page