CMU 17-648 Engineering Data Intensive Scalable Systems

Project Task 6: Scalability, Latency, and Compute Cost

Matthew Bass

Problem Description

In this project we are going to learn more about making tradeoffs between compute cost, scalability, and latency. We are going to test the response time and efficiency of our system under various loads. The goal is to increase the load that the system can handle while managing costs at the same time.

This project is focused on applying some of the techniques that we have been learning about over the last few weeks and evaluating the results.

Learning Objectives

This project will allow you to become familiar with some of the approaches that support scalability and impact response time. We will continue to evaluate our system with respect to various performance measures.

We will work with the technologies that we have been working with over the last couple of projects.

These technologies include:

- Javascript
- Node JS
- Express
- MySQL/Dynamo/NoSQL options
- Amazon Web Services
- Apache JMeter

You will need to make sure that your system is operating correctly and is resilient to malformed input, empty values, and so forth. In this project we will use automated testing, and if the tests fail we will not be able to get the performance data that we need.

Tasks

The functionality and overall goals of the system are the same as project 5. We will now be using an auto grading system that will compare the results you've achieved with the results your classmates have been able to achieve. Depending on your findings from the last project you'll look at the critical areas identified and figure out how you'll modify these areas in order to improve the scalability without adversely impacting the latency of the system while paying attention to the overall compute cost (you may want to consider some experimentation to determine the best alternatives). There are several kind of options that you'll want to consider to support this:

- Refactoring the data model: You might want to consider partitioning the data to allow for data models that are optimized for the expected load. The criteria for considering this should be the anticipated load (i.e. is the load inherently different for various kinds of data)
- Different database technology: If you found that MySQL was a bottleneck on your system one option is to utilize a different more scalable solution. You'll need to balance this against the option of going with a larger more powerful node and staying with MySQL
- Auto scaling group: If you find that the data tier is able to scale but your beginning to have trouble on the front end you can scale the number of nodes that perform the work.
- State management: Pay attention to how you are managing session state. Be aware of what happens to the state as you begin to scale the server tier horizontally.

Your system will continue to support the ability to purchase products. This will mean that you will need to track the inventory of the products and ensure that you don't sell more items than you have in stock. You will initialize your system with all of the products having 5 items in stock. You can assume that the user will reside in the USA (you don't need to worry about international addresses). You can reuse the product data and script from project 4. In addition the system will keep track of the placed orders. It will allow you to get a report of the orders placed.

Your goal is to build an efficient scalable system. This means that when you consider alternatives you'll need to compare the added compute costs with the result. You will repeat the tests that you executed previously. You will need to distribute the testing infrastructure to AWS in order to achieve the needed load on the system. You should design the system to scale. The system will be tested with between 5000 – 10,000 concurrent users.

Along with this project you are given a test plan for JMeter. You might need to modify your system slightly to accommodate the test plan. If you have trouble or questions about issues you are experiencing be sure to ask early (as this is not the point of the project). Along with the test plan you are being given a data set that contains 5000 accounts that you can use to generate load. More details about how to do this can be found in the "getting started" section.

You will also be submitting a billing report. You can see instructions for how to set this up in the "getting started" section.

The return values for all of the interfaces are in JSON format. For any "post" request, the parameters should be passed as part of the body, not as a URL parameter. The structure of the specification below is in the form "***name***:value" where the name is in bold italics and the optional values are listed as sub-bullets to the "***name***". The requirements for the system are:

- **Name**: Register Users (does not require authentication):
  - **Description:** This allows new users to create an account. The result of this will be that customer information is in the system and the user will now have an account that will allow them to log on as a customer.
  - **Interface:** {BASEURI}/registerUser
  - **HTTP Method**: Post
  - **Input parameters**:
    - fname: first name
    - lname: last name
    - address: physical address of the user
    - city: city where the user lives
    - state: two letter code for US state
    - zip: 5 digit zip code
    - email: email address for the user (you can assume that if the format is correct the email is valid)
    - username: username for this account
    - password: password for this account
  - **Return Values**:
    - ***message***:
      - Success case "Your account has been registered"
      - Failure case "there was a problem with your registration"
        - The system should not allow duplicate registrations (the same user to register twice). For this system we can assume that the first and last name needs to be unique (no two users with the same first and last name) and that the username/password combination needs to be unique
- **Name:** Login
  - **Description:** Allows users to log into the system. This should create a session that will remain active until the user logs out or remains idle for 15 minutes. The user should be able to access any functionality that requires the user is authorized to use. You need to allow for someone to log in from a browser with an active session. If there's an active session and there is a new log in (from

    the same browser) you should end the previous session and initiate a new one. There could be multiple active sessions from multiple browsers.
  - **Interface:** {BASEURI}/login
  - **HTTP Method:** Post
  - **Input parameters:**
    - username: Username of the person attempting to login
    - password: Password of the person attempting to login
  - **Return Values:**
    - ***err_message***: "That username and password combination was not correct"
      - The system should not return an error if the user is already logged in.
    - ***menu:*** {menu item1, menu time 2, …} Menu of the features the user is authorized to utilize (if username and password is correct)
      - The form of the menu items should be the paths to the functions allowed by the user for example: /updateInfo, /logout, /viewUsers, …
    - ***sessionID***: (optional) if the user was logged in a unique session id is generated and used to identify this session in the future. If you are managing sessionID through cookies or in the http header you don't need to pass this parameter.
- **Name:** Logout
  - **Description:** If the user has an active session this method will end the session. If there is no session then there is no change of state.
  - **Interface:** {BASEURI}/logout
  - **HTTP Method:** Post
  - **Input parameters**
    - sessionID: (optional) A unique number identifying the session for this user
  - **Return values**
    - ***message:***
      - if the user is not logged in: "You are not currently logged in"
      - success: "You have been logged out"
- **Name:** Update Contact Information
  - **Description:** This method allows the user to update their own contact information
  - **Preconditions:** User is a registered user and is logged into system. The only required parameter is the sessionID. The others are optional and only updated if provided.
  - **Interface:** {BASEURI}/updateInfo
  - **HTTP Method:** Post
  - **Input parameters:**
    - sessionID: (optional) An ID that uniquely identifies the current session associated with the user that is updating their information. If you are managing sessionID through cookies or in the http header you don't need to pass this parameter.
    - fname: first name
    - lname: last name
    - address: physical address of the user
    - city: city where the user lives
    - state: two letter code for US state
    - zip: 5 digit zip code
    - email: email address for the user (you can assume that if the format is correct the email is valid)
    - username: username for this account
    - password: password for this account
  - **Return values:**
    - ***message:***
      - if there was a problem: "There was a problem with this action"
      - success: "Your information has been updated"
- **Name:** Modify Products (only an admin can modify a product):
  - **Description:** This method allows you to modify the description and name of the product
  - **Preconditions:** The user is an admin and is logged into the system

    - **Interface:** {BASEURI}/modifyProduct
    - **HTTP Method:** Post
    - **Input parameters:**
        - sessionID: (optional) An ID that uniquely identifies the current session. If you are managing sessionID through cookies or in the http header you don't need to pass this parameter.
        - productId: a unique number identifying the product
        - productDescription: a description of the product
        - productTitle: a short title for the product
    - **Return values:**
        - ***message:***
            - if there was a problem: "There was a problem with this action"
            - success: "The product information has been updated"
- **Name:** View Users
    - **Description:** This method allows the user to view all of the users registered in the system
    - **Preconditions:** The usre is an admin and is logged into the system
    - **Interface:** {BASEURL}/viewUsers
    - **HTTP Method:** Get
    - **Input parameters (optional parameter to filter results):**
        - sessionID: (optional) An ID that uniquely identifies the current session. If you are managing sessionID through cookies or in the http header you don't need to pass this parameter.
        - fname: some portion (or all) of the first name of the users you'd like to view
        - lname: some portion (or all) of the last name of the users you'd like to view
    - **Return values:**
        - ***user_list:***
            - {user1, user2, …} List of users meeting criteria provided
            - User list should be firstname lastname. In the future you could imagine having a list of firstname lastname that is a hyperlink that will show you all of the details of the user if selected (not required for this system).
- **Name:** View Products (does not require log in):
    - **Description:** This method returns products that meet the search criteria
    - **Interface:** {BASEURI}/getProducts
    - **HTTP Method:** Get
    - **Input parameters (optional)**
        - productId: Id for a specific product that you'd like to view
        - category: category for the products that you'd like to retrieve
        - keyword: a keyword that can be used as a search term in the title or product description.
    - **Return values**
        - ***product_list:***
            - {product1, product2, product3, …} List of products that meet the search criteria
            - The list should be the product title. As with the users you could imagine having each be a link for the details (in the future, not now).
- **Name:** Buy Product (requires user to be logged in)
    - **Description:** This method simulates the purchase of a product. It will be successful only if the user is logged in and if there is a product available.
    - **Interface:** {BASEURI}/buyProduct
    - **HTTP Method:** Post
    - **Input parameters:**
        - productId: Id for a specific product that you'd like to purchase
        - sessionID: (optional) An ID that uniquely identifies the current session. If you are managing sessionID through cookies or in the http header you don't need to pass this parameter.
    - **Return Value:**
        - message: 01 the purchase has been made successfully

02 you need to log in prior to buying a product
03 that product is out of stock

- **Name:** Get Orders (requires user to be logged in as admin)
  - **Description:** This method generates a report that gives the rank of the products sold
  - **Interface:** {BASEURI}/getOrders
  - **Http Method:** Post
  - **Input Parameters**
    - sessionID: (optional) An ID that uniquely identifies the current session. If you are managing sessionID through cookies or in the http header you don't need to pass this parameter.
  - **Return Value:**
    - Array of [productId: id, quantitySold: number sold]
    - message: 01 the request was successful
      02 you need to log in as an admin prior to making the request

Product Data:

- A separate data file will be provided for the product related information.
- A product contains:
  - Product ID: Id of the product
  - ASIN: Amazon Standard Identification Number
  - Title: name of the product
  - Description: description of the product (current data set this field is blank, but we need to be able to add descriptions)
  - Group: Book, Music, DVD, or Video
  - Category: Location in product category hierarchy to which the product belongs (separated by |, category id in [])
  - Price: price of the item

You should assume that multiple people will be accessing your system simultaneously and be sure that your system operates correctly when you have concurrent users. You are expected to use good programming practices with proper error handling. You will be given a set of users to load in your database prior to submitting your system.

You will build the web application using Node JS and Express with your choice of database technology. The system will be deployed and tested on AWS. You will have a budget of $10.00 for this project that includes development and testing.

Technologies and Resources

You will be using Javascript, Node js, Express, and Amazon Web Services (AWS) for this project. Links for general descriptions and resources are listed below. In the next section we will give detailed instructions on installing the technologies, getting started with AWS, and deploying an application to AWS.

General links:

- Javascript: http://www.w3schools.com/js/
- NodeJS: https://nodejs.org/
- Express: http://expressjs.com/
- AWS: http://aws.amazon.com/
- Apache JMeter: http://jmeter.apache.org/

Getting Started

Before doing anything more you will need to ensure that CloudTrail is still set up.

If you've not yet set it up you'll need to follow the steps below:

- When you sign into AWS ensure that the East US Region is selected (N. Virginia)
  - You'll see the currently selected region listed in the middle drop down on the top right hand side of the page.
  - To change you select the desired region from the drop down list
- From the main AWS console select "CloudTrail"
  - At any point you can return to the AWS console by selecting the Amazon icon on the top left hand corner of the page
- From the opening CloudTrail page select "Get Started"
- You should see two text boxes and one set of radio buttons
  - For "Create new S3 bucket" select "no"
  - For the name enter "*andrewID_logs"*
  - For "S3 bucket" enter "ediss-project-logs"
- Select "Turn On"
  - AWS will check to make sure you have permissions to write to this bucket. If all is well it will accept the data entered.
  - If there is an issue you'll get a message next to the S3 bucket text box.
    - Be sure you've entered the bucket name correctly and
    - Double check to make sure you've sent the correct Amazon ID to the instructor
    - If both of these have been done correctly contact the instructor
- AWS will now log your activity to the class S3 bucket

Setting up AWS:

Once you have an AWS account and have your developed and tested your application locally you'll need to deploy your application to the cloud. This is a two-step process. You'll need to set up an instance to run your node application and you'll need to set up your database.

- Node instance: Starting an instance on AWS is pretty easy. The directions for setting up an instance can be found here http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-launch-instance_linux.html. Follow the instructions to launch the instance.
  - When you create an instance it is deployed into a Virtual Private Cloud (VPC). This is essentially a virtual firewall. As with other firewalls you'll need to create rules that will allow specific kinds of access in order to be able to connect to your instance.
  - You'll want to be sure that you can connect to your instance via port 22 in order to connect with an SSH client. You can specify a specific IP address where you'll be accessing your system from or a range of IP addresses (e.g. from your school's network). You don't want to allow unrestricted access, however, as this increases the likelihood that your account can be compromised. You can read more about setting up access rules at: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html#vpc-security-groups. In addition to opening access for an SSH connection, you'll want to grant access from anywhere for port 3000 (the port that node js uses). Otherwise you won't be able to access your node application.
  - Once you create an instance on AWS you'll need to install node & express as you did on your local machine. You can connect to your instance with SSH. You can see directions here http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-connect-to-instance-linux.html.

- Database Instance: You'll now need to create a database instance on AWS. You can find directions here http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_CreateInstance.html.
  - Once you create an instance you'll need to once again create rules to be able to connect to the database. You'll want to allow access from your machine in order to be able to configure the database. Additionally, you'll need to allow access from the EC2 instance that you created, otherwise your node application will not be able to access the database.
- Once you have the instances set up deploying your application is a matter of moving your node project to the EC2 instance and uploading your database to your db instance. Be sure to change your code so that your application is accessing the database running on AWS and not on your local machine.

Running JMeter:

- In order to do load testing you'll need to run JMeter remotely via the command line. You can find information about how to do this at https://blazemeter.com/blog/dear-abby-blazemeter-how-do-i-run-jmeter-non-gui-mode or http://jmeter.apache.org/usermanual/get-started.html#override. The test plan that you are provided expects the following parameters to be passed: IP address (host.ip), Port # (host.port), and the filename for the output (filename).
- The –J flag indicates that you have a user parameter that you are passing. The –n flag means run without the GUI, and the –t tells JMeter what test plan to run. Here's an example command to execute JMeter with this test plan via the command line:
  - **>C:\JMeter\apache-jmeter-2.13\bin\jmeter -n -tProject6.jmx -Jhost.ip=localhost -Jhost.port=3000 -Jfilename=mattbass**

Billing Report:

- You'll need to submit a billing report that details the usage for the grading as part of your submission. In order to do this you'll need to set up your system appropriately. The steps you'll need to follow are:
  - Tag all of the instances in your system with Project: Project 6. In order to do this you'll go to the AWS console for all of the instances (EC2, RDS, Dynamo, Redis, …) and select "add tags". You then add a tag with a Key = "Project" and a Value = "Project 6"
  - Create an S3 bucket if you don't already have one (*Creating a Bucket*)
  - Enable detailed billing reports (*http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/usage-reports.html#usage-reports-prereqs)*
  - Create a detailed billing report for the **period when the project was submitted.** **http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/usage-reports-instance.html.** Be sure to select hourly granularity and submit both a .csv file as well as the graph in .png format.

Submission

For this project you'll submit:

- *The code*
- *The billing report for the period when you've submitted the system for grading (as described in "getting started")*
- *Submit your system to the auto grader. The IP address for the autograder is* 52.4.234.34:3000

Grading

The grading for this project will be:

- Your project will be evaluated on 4 dimensions:
  - Functional correctness (should gracefully handle incorrect input)
  - Scalability
  - Response time
  - Compute cost

- For each of these dimensions you'll be ranked alongside the others in the class. 60% of your grade will be how you perform with respect to the others in the class. 40% will be given to you for being able to build a system that can correctly handle 1000 concurrent users.
- The grading criteria will be as follows for each dimension:
    - Top 1/3 of the class 20/20
    - Middle 1/3 of the class 15/20
    - Bottom 1/3 of the class 10/20
- You must submit your system to the auto grader prior to the due date/time. The auto grader can be found at 52.4.234.34:3000. You can submit your system as many times as you'd like prior to the deadline. You can only submit two times without any penalty. For every submission over 2 you will get a 5% overall penalty.
- **You must have enabled CloudTrail logging BEFORE starting this project in order to get credit**

Budget

The budget for this project is $10.00 (or resources that would total $10 without the free tier). Let me know if you've exceeded your past budgets and need more credits.