QuestMaster: A Platform for Posting and Performing Real Life Quests

CS290B: Scalable Internet Services (Fall 2014)

Bay-Yuan Hsu

Chad Simmons

Karthik Puthraya

Yalun Qin

1. Introduction

Questmaster is a web application where the users can perform real-life "quests" for other users in return for rewards. The quests are localized to geographic locations. A user can create quests for others to complete in return for rewards which can be monetary or non-monetary. Similarly, a user can accept quests from other users and collect rewards. These collected rewards may be redeemed at any later point in time. Main features of our web application are:

- Create a new quest with location information
- Browse the quests and view their contents
- Search for quests by title, location or any other attributes
- Accept and complete a quest

Questmaster is implemented in Ruby-on-Rails and uses MySQL as the backend data store. In the following we will discuss about the experience and lessons we have learned during the development of Questmaster and show the load testing result.

2. Development Model

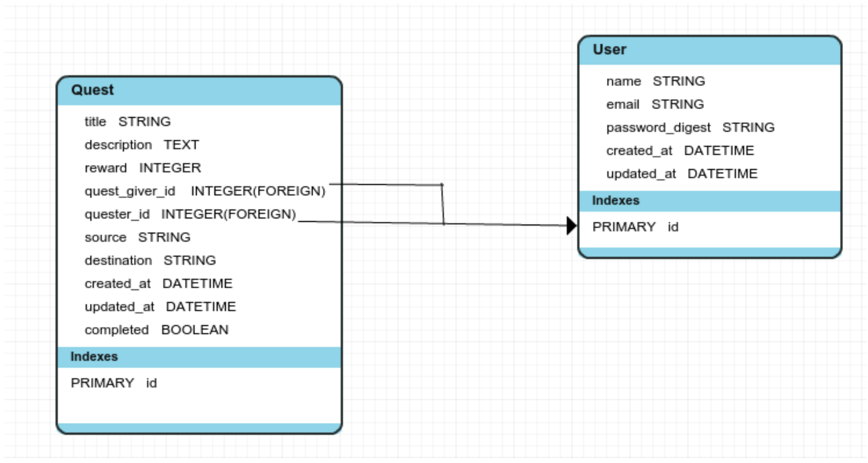
The development model we used is agile model. The whole development process contains several sprints, each of which lasts about one week. At the end of each sprint all our team members had a meetup, reviewing things that had been done in the last sprint, discussing about issues we had encountered and making a plan for the next sprint. We also demonstrated our accomplished work in the last sprint to the instructors and got feedback from them. To make sure all of us know the work of each other, we used PivotalTracker to post the features to be added and bugs to be fixed. Having a clear goal in mind, each team member tried his best to complete his work and prepare for the next meetup.

To make our cooperation more efficient, we hosted our code on GitHub (https://github.com/scalableinternetservices/Motley-Crew) and used Git as the version control tool. Each of us worked on a separate branch and after it has been thoroughly tested, he makes a pull request and it is merged back into master after other members are satisfied with the changes.

3. Application Overview

As with other rails applications, our web application follows the model-view-controller(MVC) architectural pattern, which separates business logic from the front-end logic associated with GUI. To make it easier for us to beautify the web pages, we integrated Twitter Bootstrap framework into our application and used lots of css styles and javascripts provided in that library. In this way, we are able to focus more on the logic and model side, which is more interesting to us.

One critical thing about our application is the design of models. For now we are using two models--User and Quest, each of which corresponds to a table in the database. Since each user can accept or post multiple quests and each quest can only be given or accepted by one user, we designed the quest model to have two foreign keys, which are the indices of quest giver and quester. This can be illustrated by the following figure.



Ideally, we should consider two kinds of quest - quest that can be accepted by one person and quest that should be performed by more people. Due to the time constraint we are unable to support the second type since that requires adding one more intermediate table specifying the role of each user for each quest.

4. Scalability Testing

After the initial development, we hosted Questmaster on Amazon EC2. We did all our testing using the templates provided to us. So, our production environment consisted of the following:

- Web server: Nginx
- Application server: Phusion Passenger
- Database: MySQL server

We did all our load testing using Funkload. We used the distributed framework of Funkload with 4 worker m3.2xlarge EC2 instances. We wanted to ensure that the load testing is not throttled by the network bandwidth or the CPU resources of the load-testing clients.

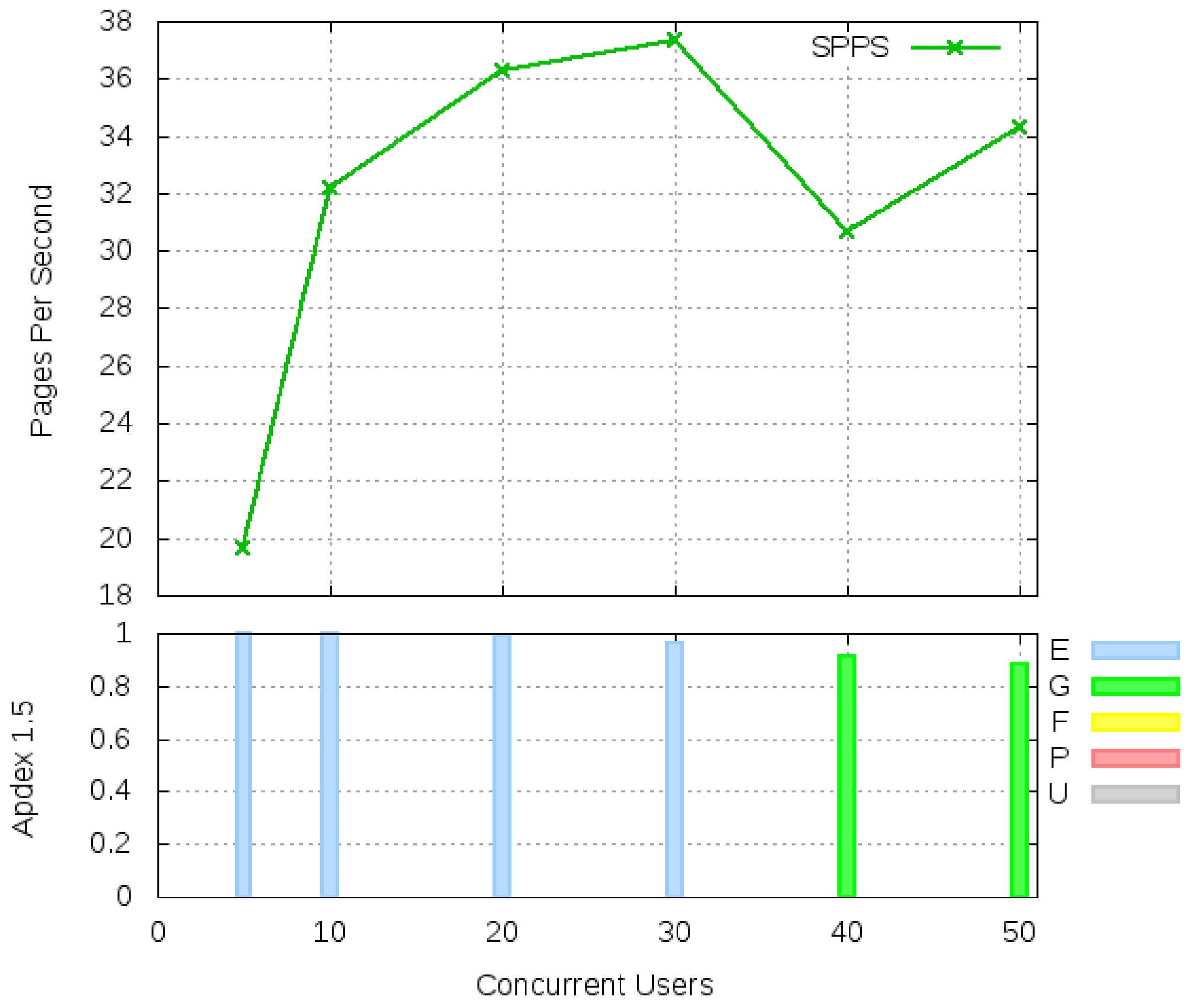
All our testing started with a test dataset consisting of about 15,000 users. Our main testing critical path consists of the following operations:

- GET the root url
- GET sign-up url
- Generate random user credentials and POST form corresponding to the new user
- Upon successfully creating, GET the new user's profile page
- Log into the server, GET new-quest page to create a new user quest
- Generate random quest details and POST the quest form to the server
- Upon successfully creating the quest, GET the user's profile page
- Logout of the server by sending a GET request

The whole critical path takes about 2-3 seconds to execute in a typical run of the Funkload unit test.

4.1 Initial Test

We started our testing runs with a single instance deployment on a t1.micro EC2 server. We noticed fair loading times in a browser. However, when we subjected the instance to Funkload's test suite, the app broke with very little load, as expected. Here are the results of the very first test we ran:



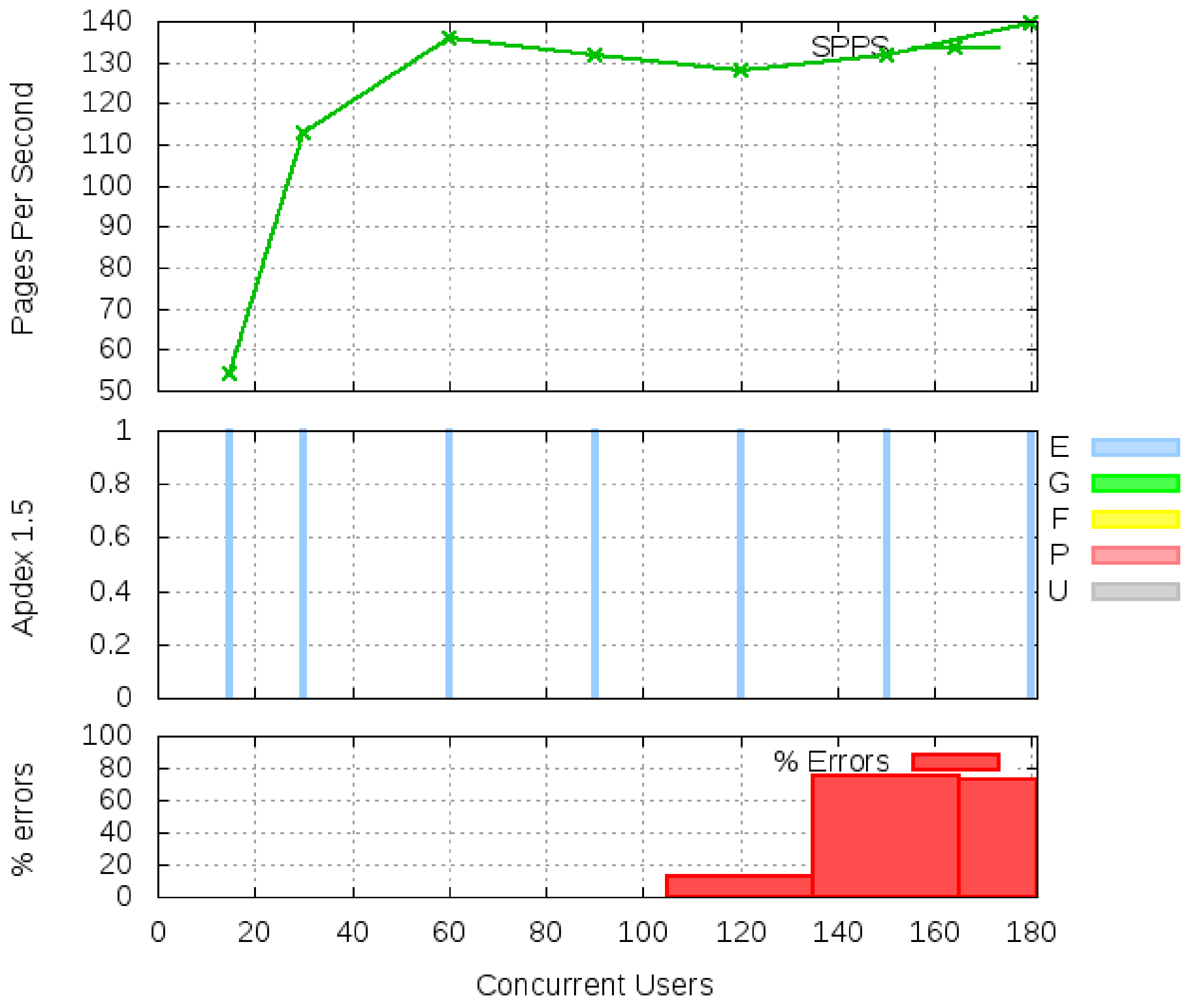
As can be seen, the performance starts to deteriorate even with just 40 concurrent users. We need to scale up and out to see better performance.

4.2 Scaling Up

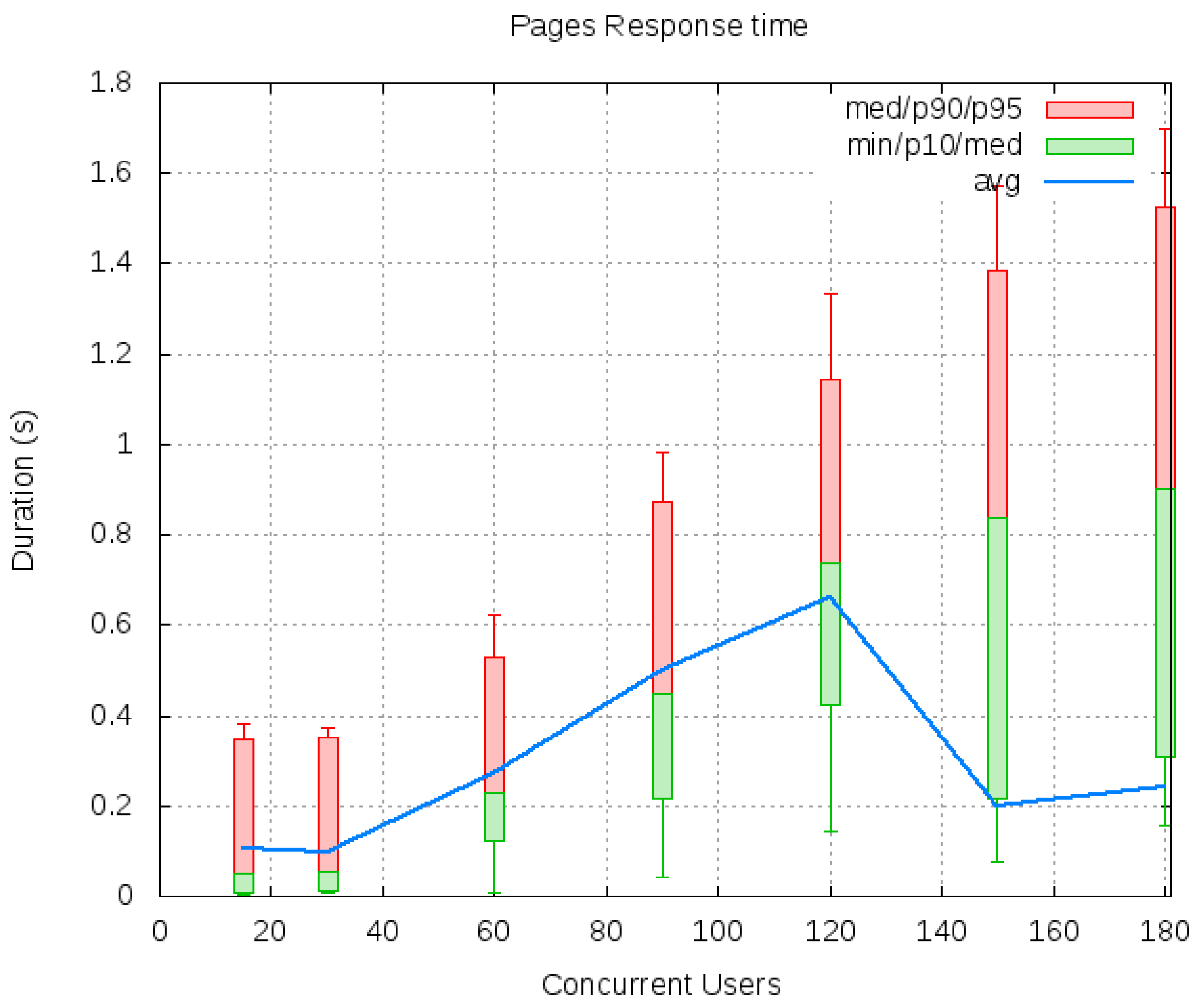
Our next step was to scale up by using bigger instances of the app server. The database is still hosted on the same instance as the app server. We gradually scaled up the size of the EC2 instances and got higher rates of concurrency with each scaling up.

m3.xlarge

Here are the results of load-testing against a m3.xlarge instance



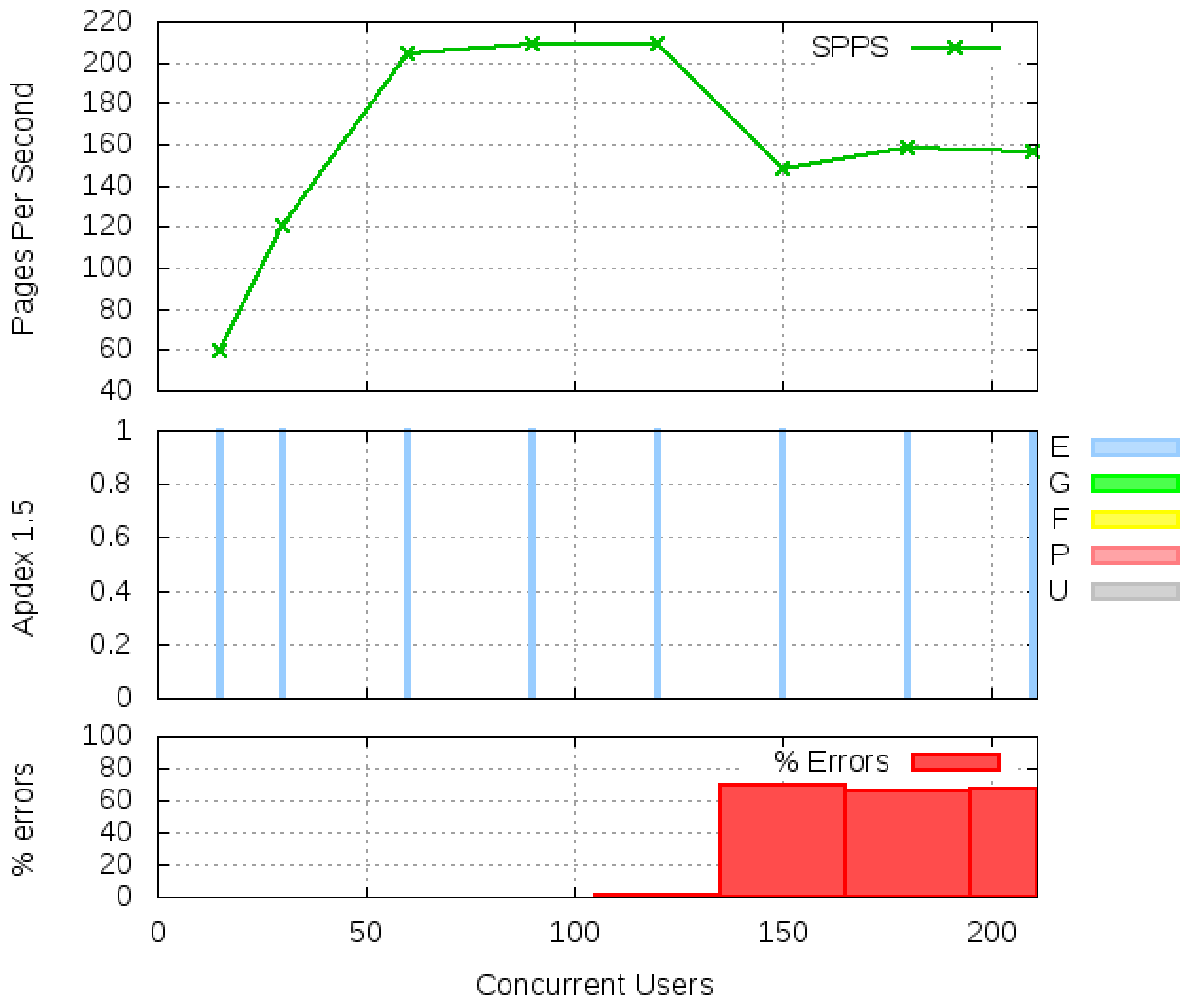
As can be see, we can serve up to about 130 pages per second to up to 100 concurrent users before we start running into errors. The following is the latency plot of the same instance.



As can be seen, the latency is below 500 ms for up to 100 concurrent users which is when we start encountering errors.

c1.xlarge

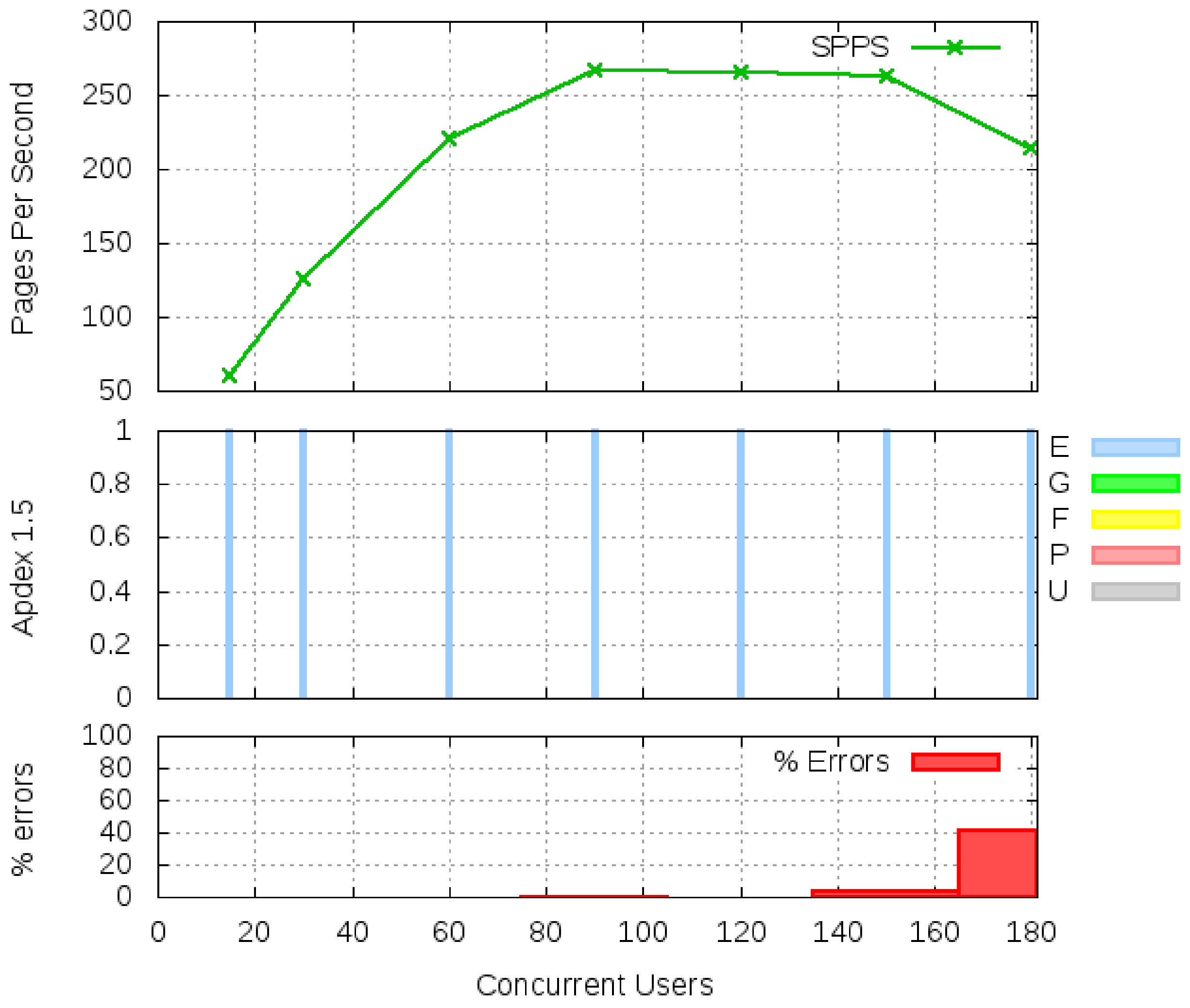
We now scaled up to a c1.xlarge instance. To our surprise, we did not find any considerable improvement over m3.xlarge.



As can be seen, the highest number of concurrent users is not 110 - marginally greater than that of m3.xlarge. However, the worst case latency is now under 400 ms.

m3.2xlarge

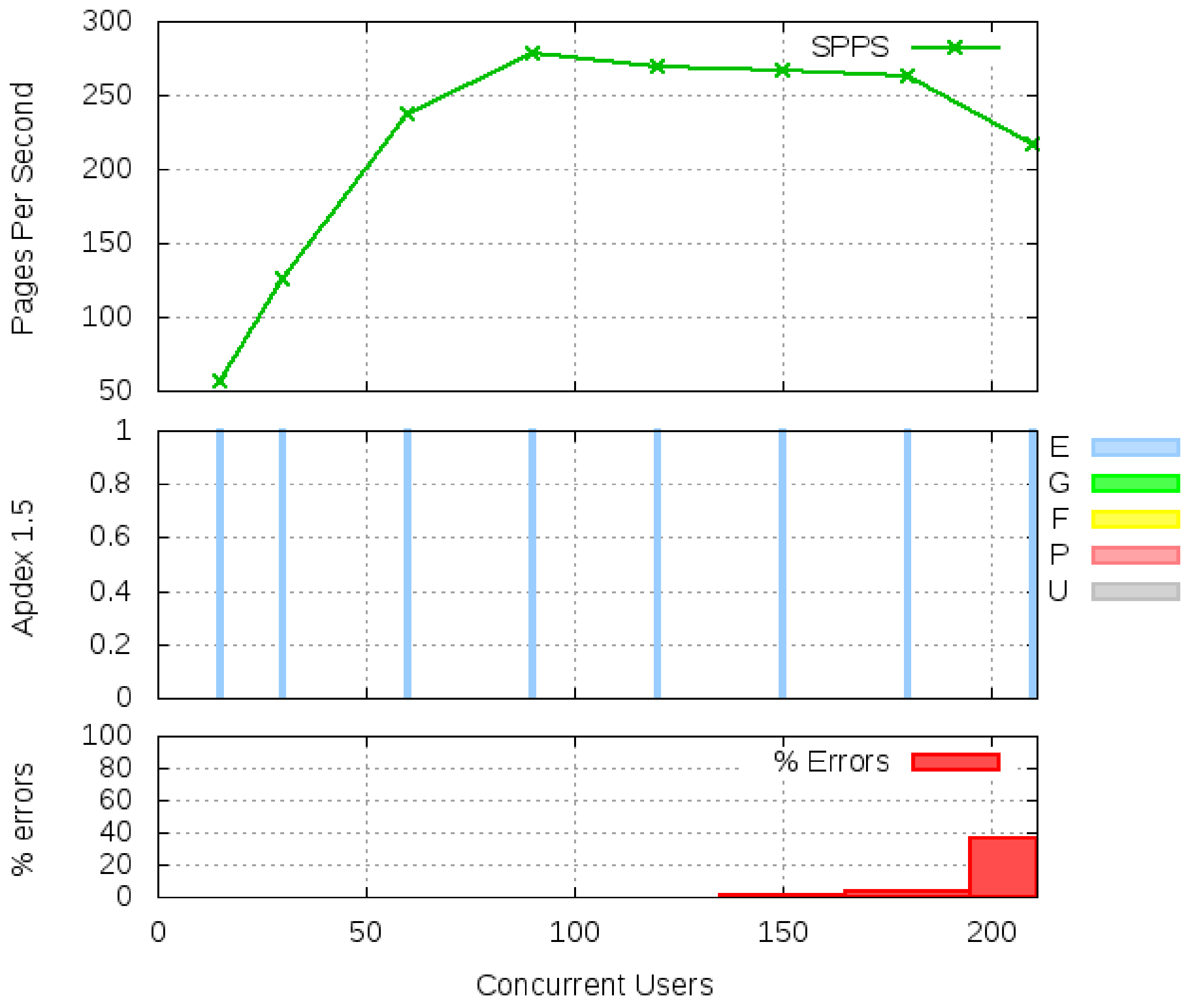
Our next step was to scale up to an m3.2xlarge single instance setup. Below is the output of the Funkload test.



As can be seen we have gained slight improvement in concurrency with the server now able to support up to 130 users. The worst latency for this case too is about the same as that of a c1.xlarge instance (~400 ms).

c3.2xlarge

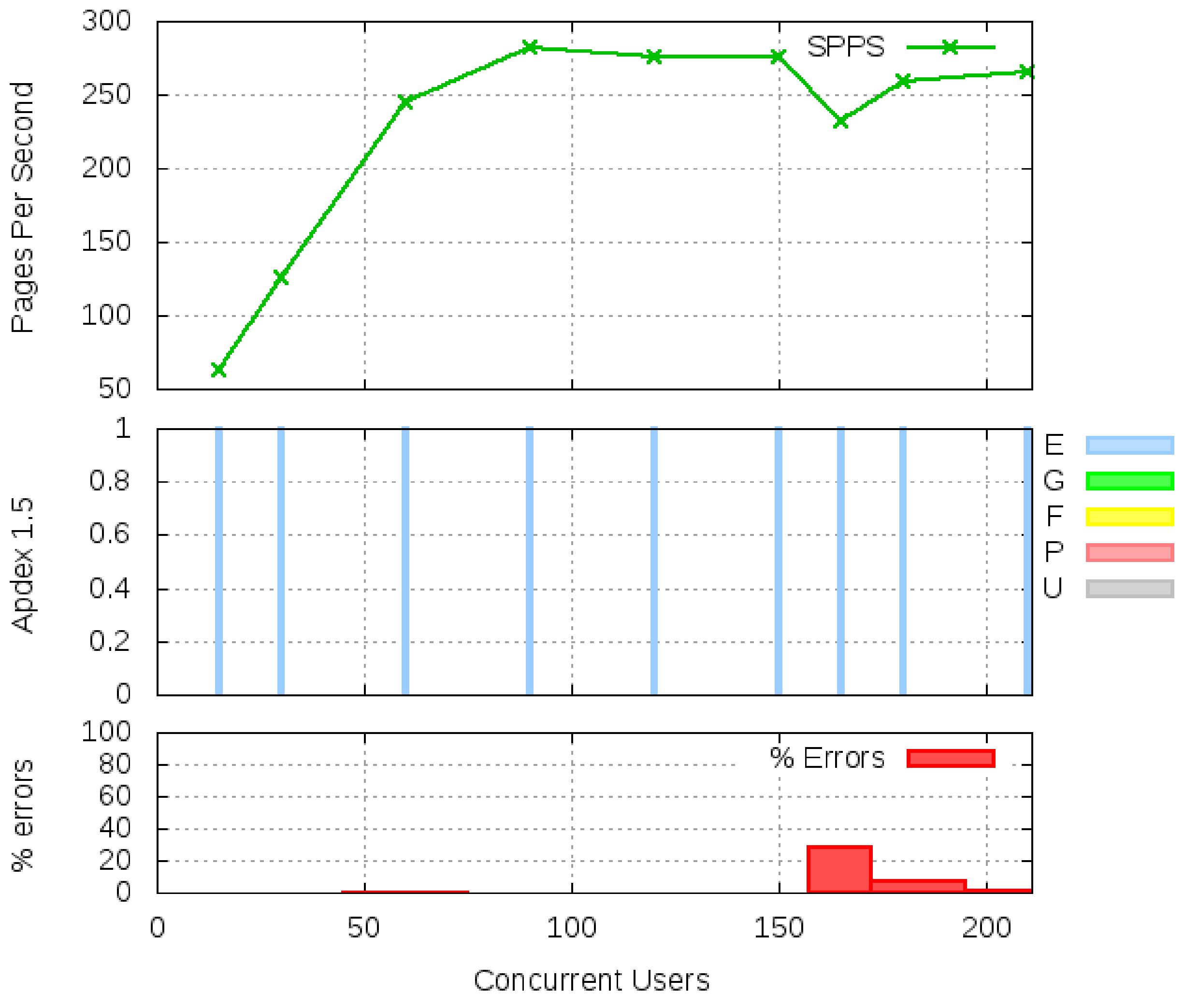
The next scaling up we did was to switch to a c3.2xlarge instance. We again gained a marginal hike in concurrency with the latency being the same.



We could now support about 140 concurrent users before we hit any errors.

c3.4xlarge

Our final step in scaling up was to use the best single instance we could get which was a c3.4xlarge EC2 instance. This setup gave us our best results with a single instance setup.



We are now able to support about 160 concurrent users with a maximum latency of 400 ms.

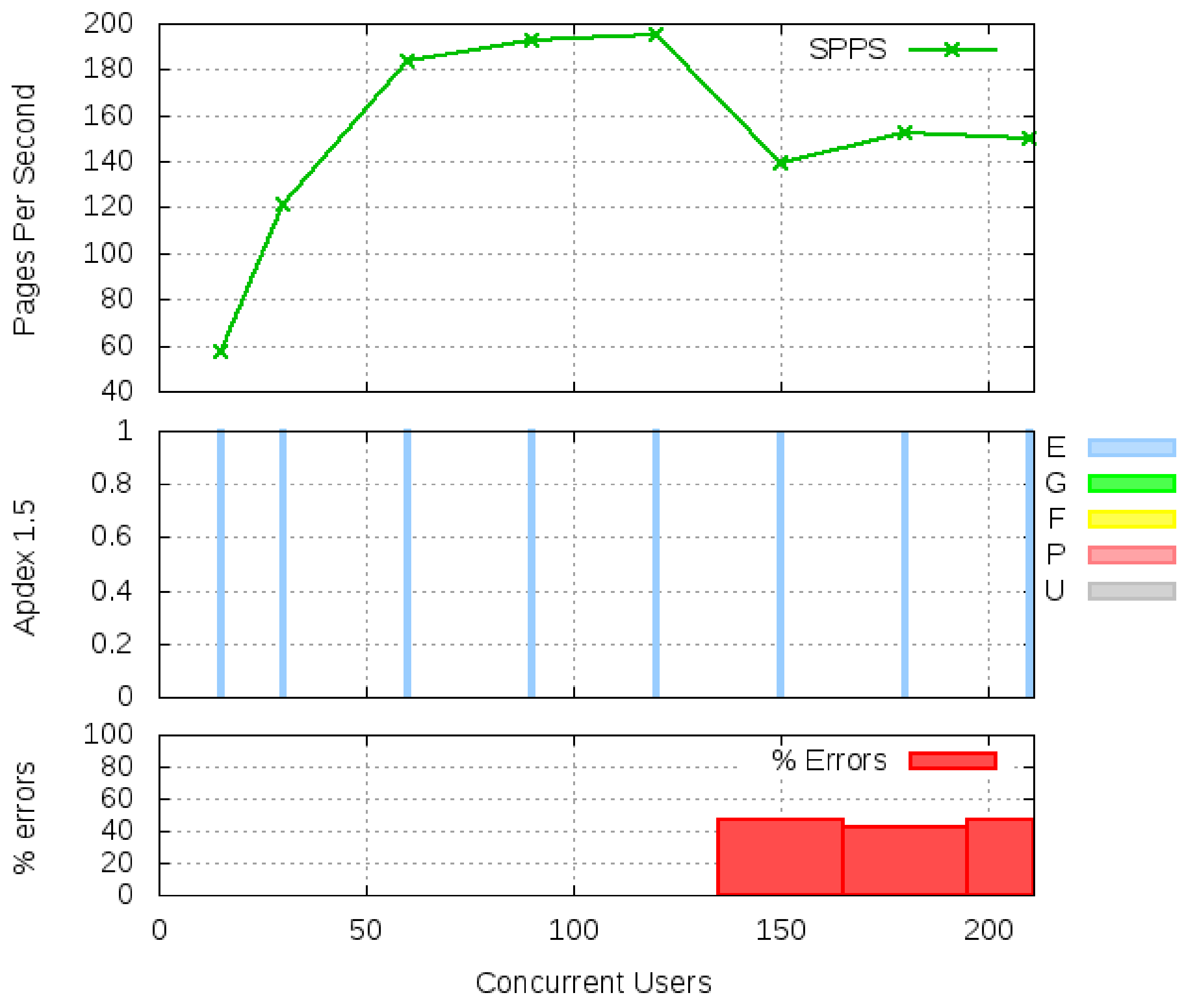
4.3 Scaling Out

The limitations of single server instances is obvious from the above study in scaling up. Even with the best available machine, we were not able to achieve a concurrency of even 200 users.

The solution to this is to “scale-out” and have multiple app-servers and web-servers serving the users. The load is distributed among the app-servers by means of a load-balancer. The SQL database is also separated from the app-server. Previously, the database used to be hosted on the same instance as the app-server/web-server. We now have a dedicated database server serving all the app-servers.

2 x m3.xlarge

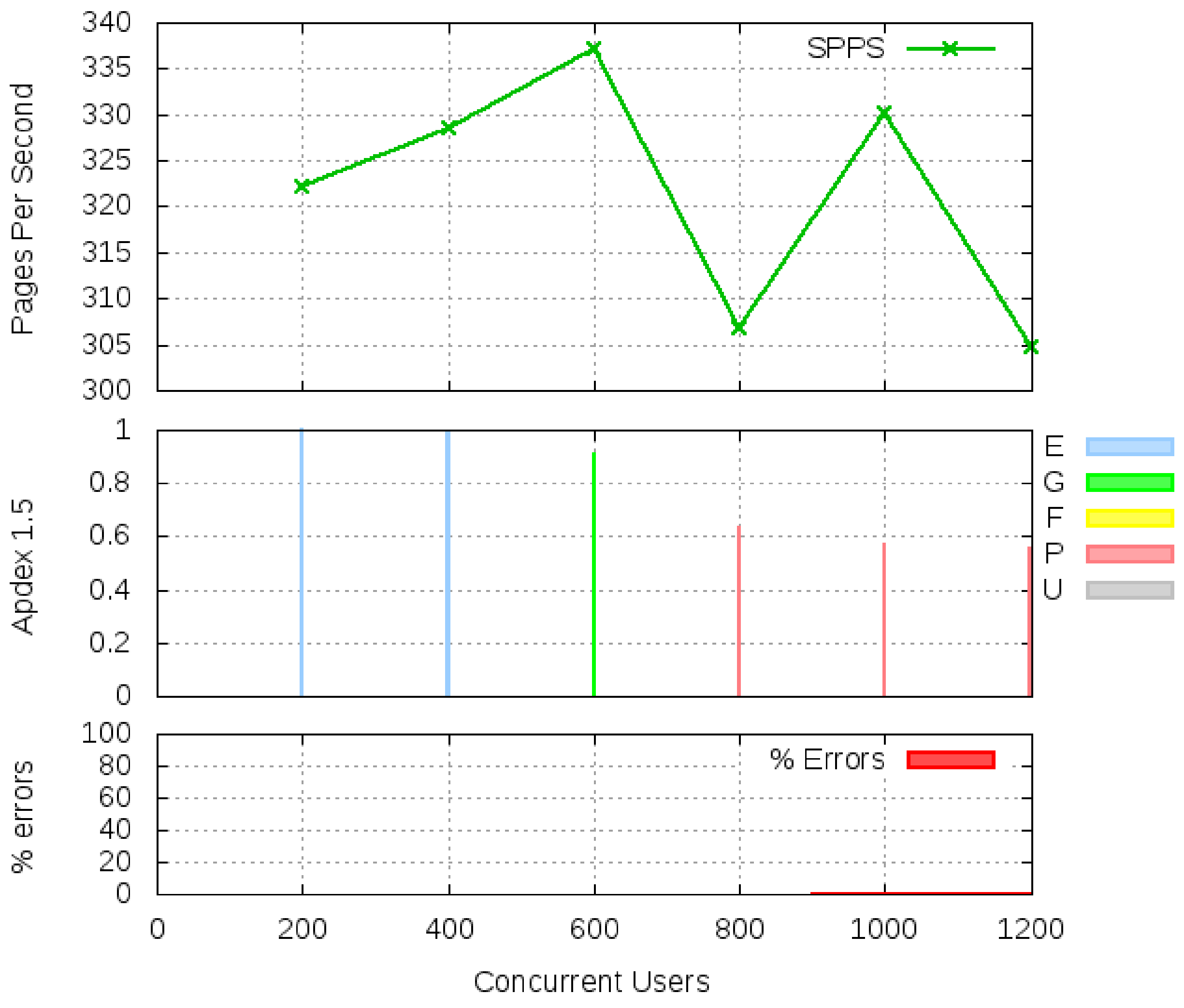
We begin our study in scaling-out with 2 app-servers, both hosted on m3.xlarge EC2 instances. The database is a of DBInstance Type db.m3.2xlarge. The following is the report generated by Funkload.



As can be seen, we achieve a maximum concurrency of 150 users before we start running into errors. Contrast this with that of a single m3.xlarge instance where we were able to achieve a concurrency of only 90 users. Although we achieved greater concurrency, we were a little disappointed to see that the scaling is not linear as expected. The worst case latency was about 450 ms.

8 x m3.xlarge

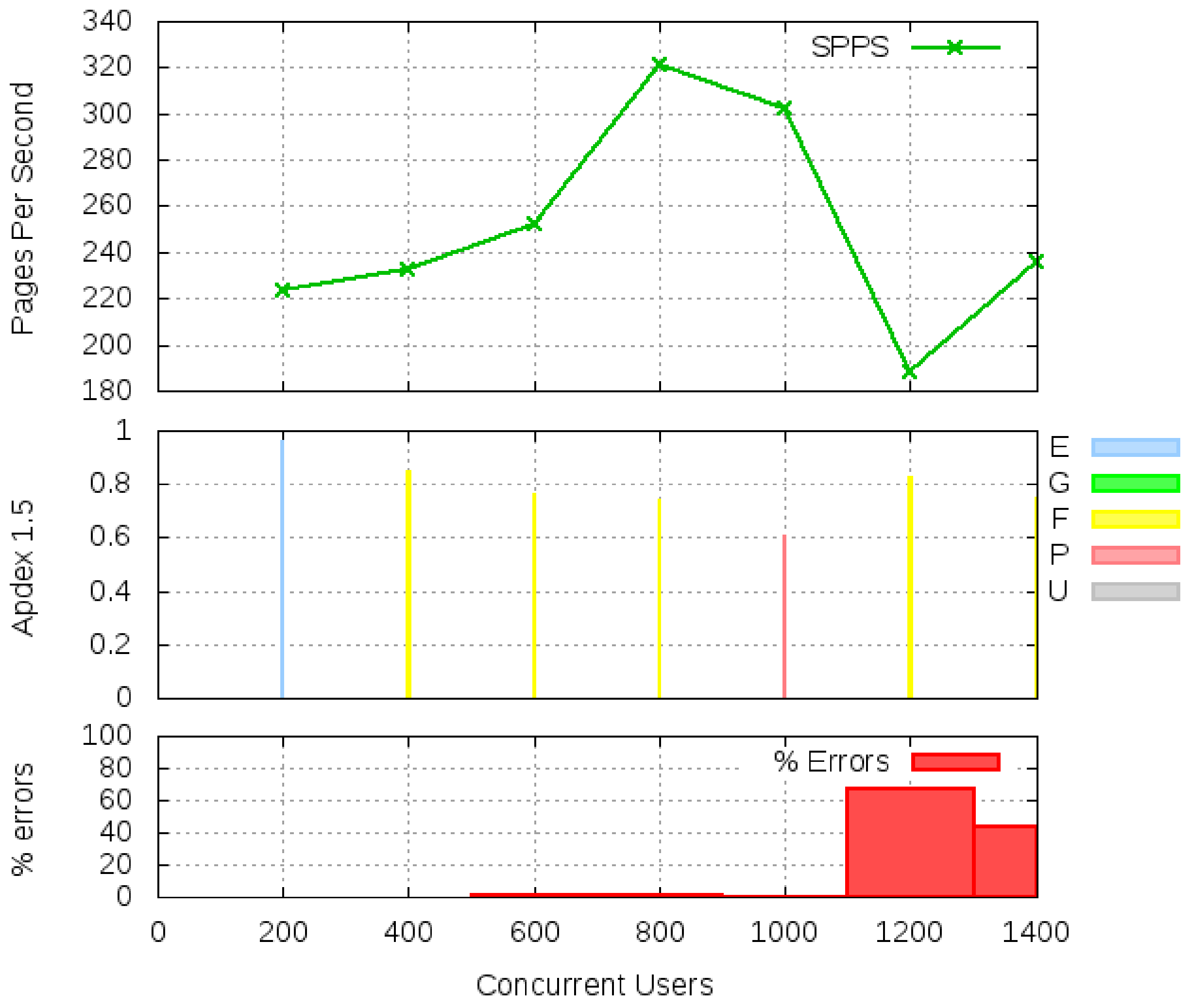
We added 6 more m3.xlarge app servers to the previous setup. The DB instance was the same. We gained the expected 4x performance with the number of concurrent users. Here are the results for the same:



As can be seen, we were able to hit 600 user concurrency which is about times the 150 users we were able to support with the 2 x m3.xlarge setup. The worst case latency at 600 users was about 1.5 sec.

4 x c3.xlarge

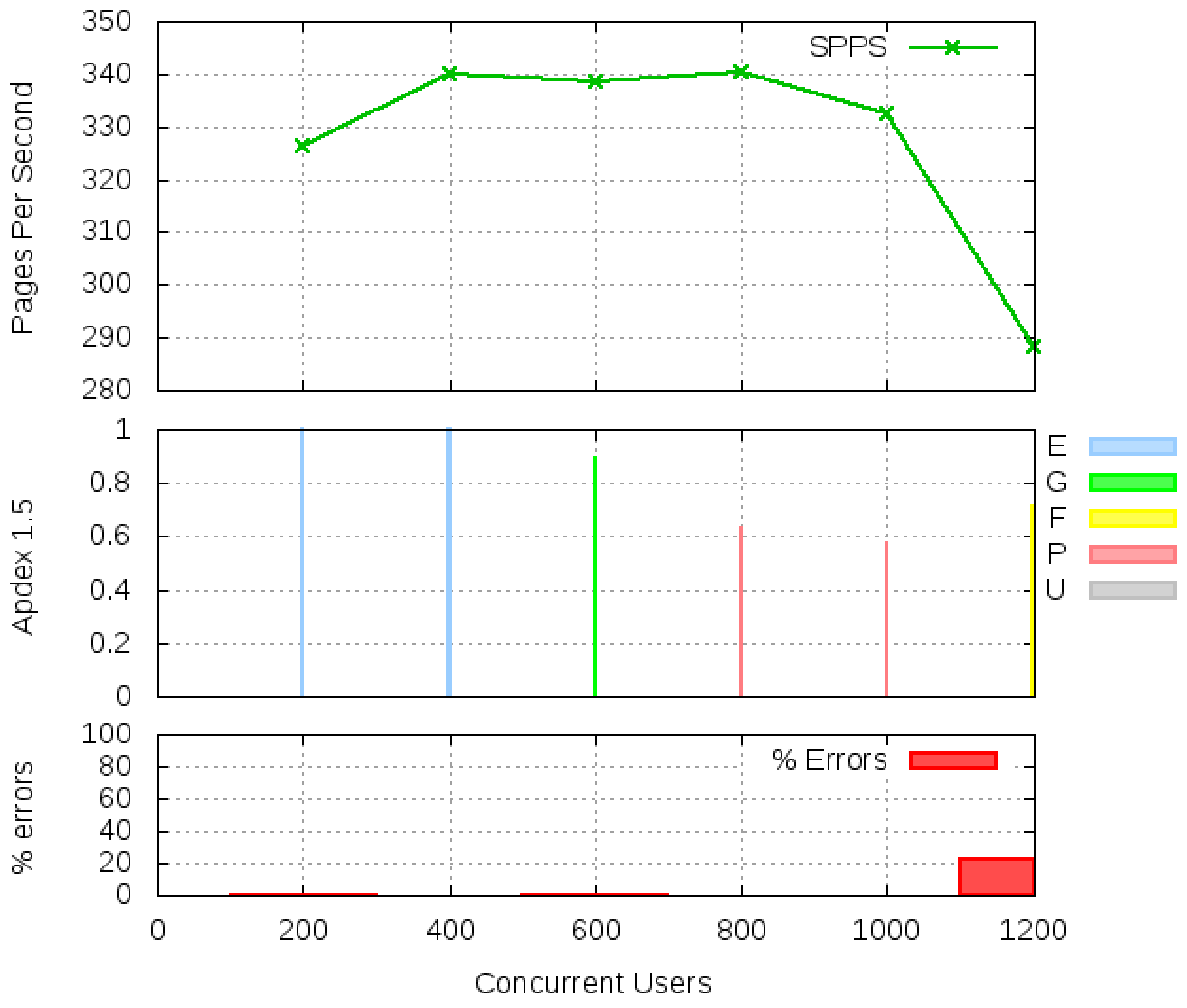
We also tried scaling our c3.xlarge single instance setup by adding multiple app servers. Here are the results of the same:



We were able to reach about 1000 concurrent users with little error. This is about 4 times the number of users we were able to support with a single instance of c3.xlarge. However, the latency exceeded 2 secs at about the 600 user mark.

8 x c3.xlarge

We now further scale horizontally and add 4 more app servers with the same instance of the database. The results are as follows:

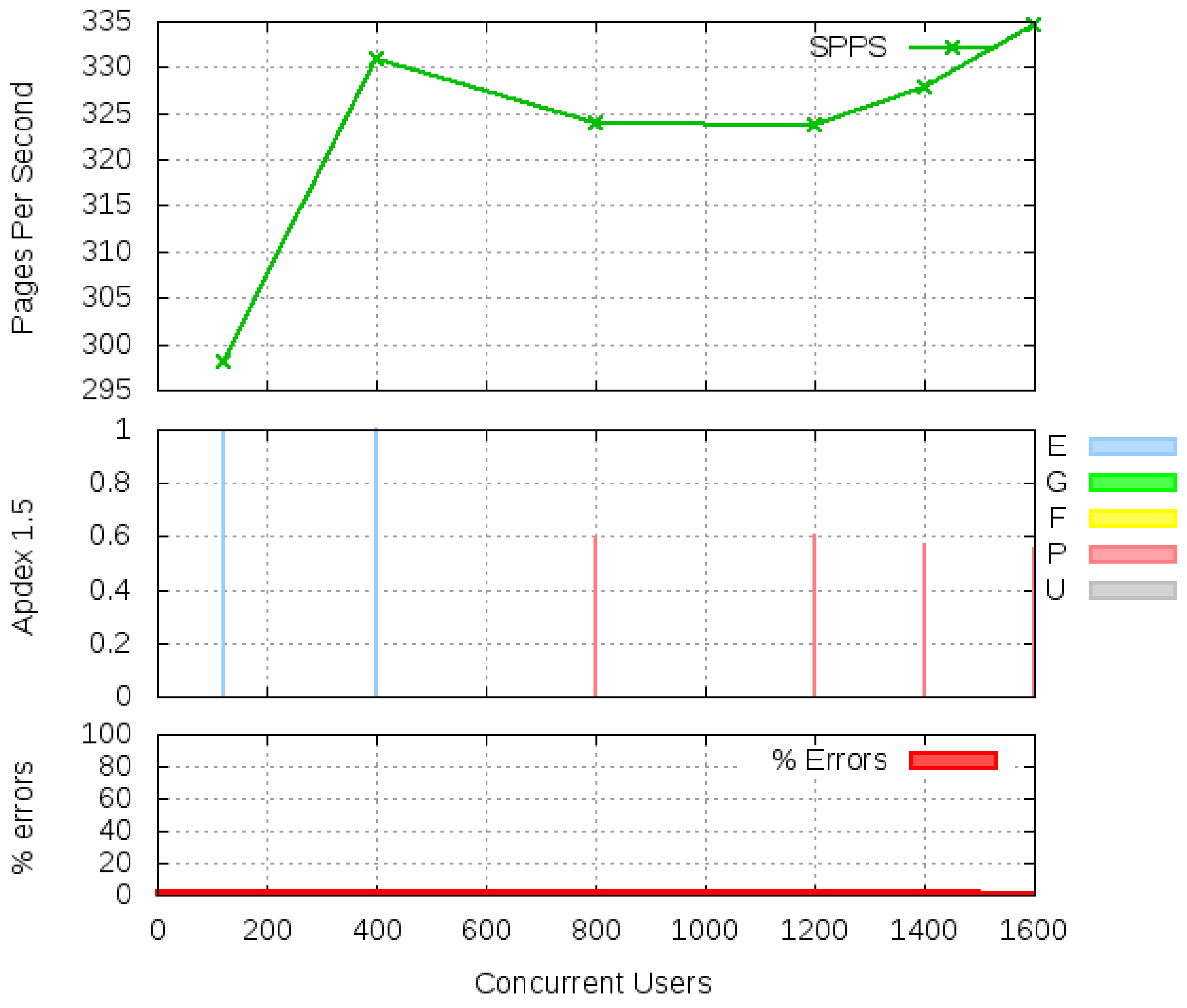


Even though we doubled the number of app servers, we did not see a doubling in the concurrency of users. We started encountering errors at the 1200 user mark whereas we expected the number of concurrent users to be somewhere in the range of 1600. Also, the latency periods were close to 2 sec at around 800.

Phusion Passenger Optimizations

8 x c3.xlarge

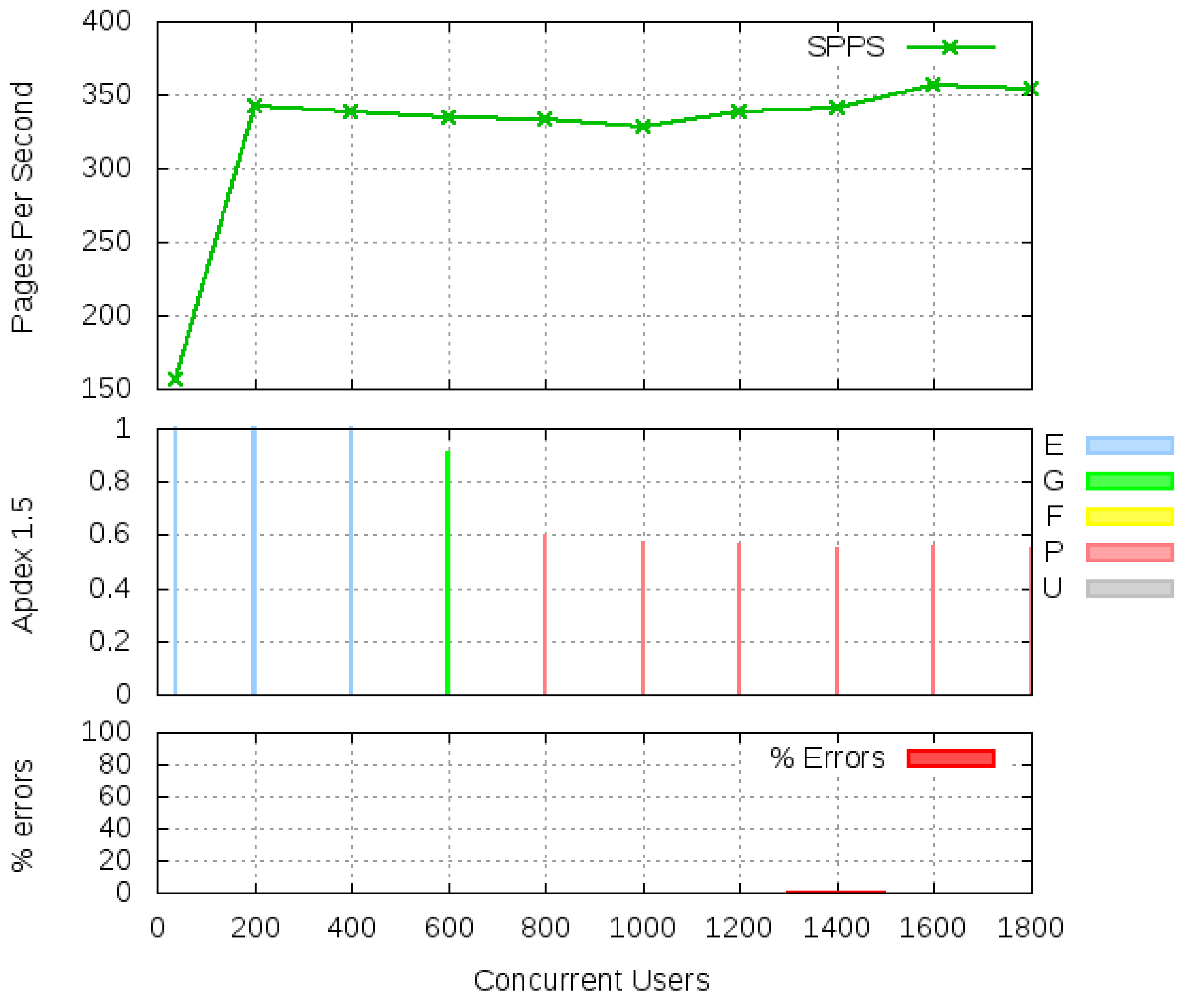
We now tried to increase the number of worker processes on the c3.xlarge EC2 instances. Since each of these VMs have 4 CPU cores, we configured Passenger to spawn 8 worker instances - 2 per core. The following is the result of the Funkload test on this setup.



We were able to hit the 1600 concurrent user mark without significant errors. However, the latency still exceeds 2 secs at around the 700 user mark. This shows that the bottleneck in our setup is the SQL server.

8 x c3.4xlarge

We finally tested our application on a 8 x c3.4xlarge setup. Here are the results for the same.

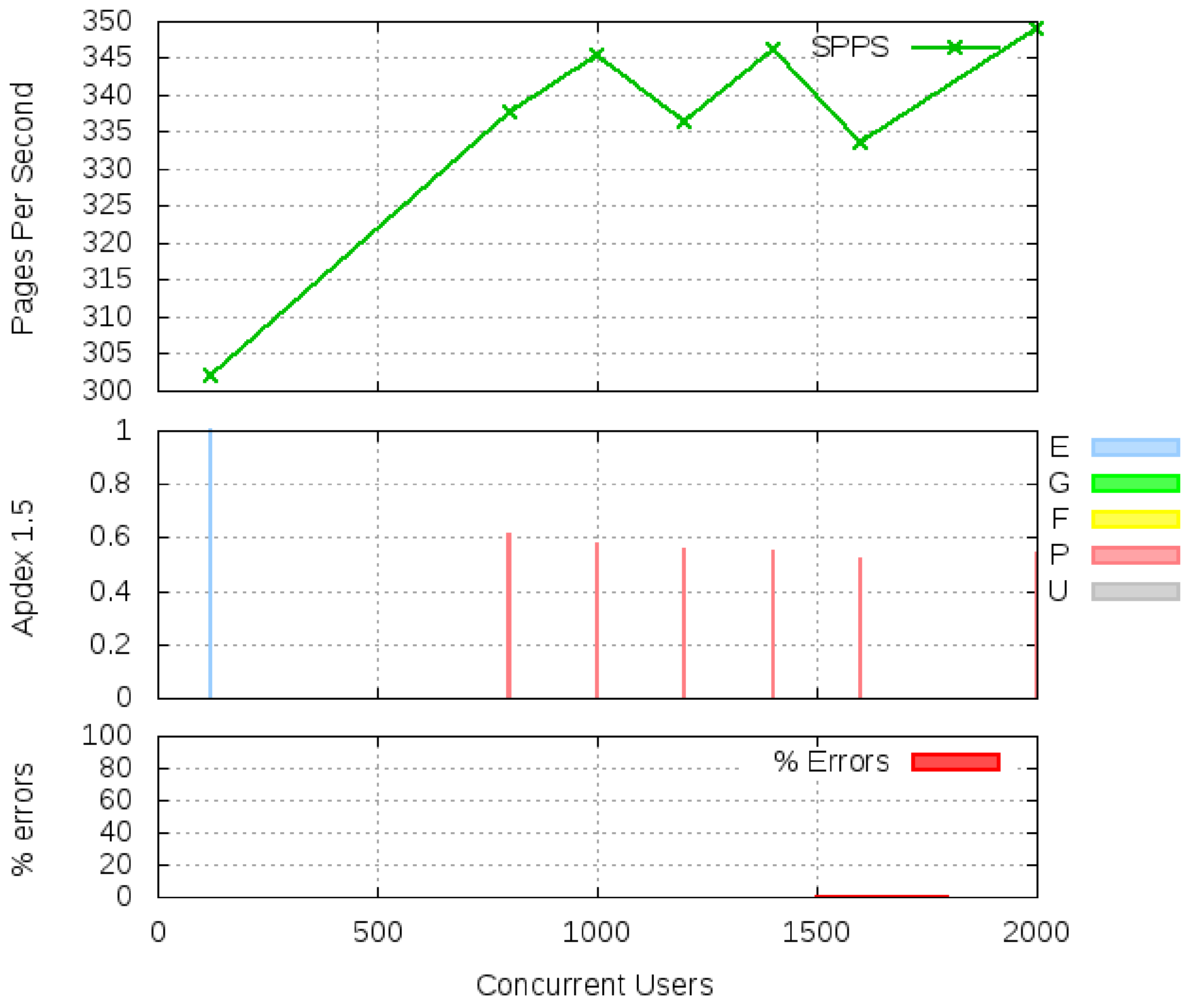


We are able to get a concurrency rate of 1800 users.

SQL Optimization

8 x c3.xlarge

The final optimization we made was to add indices to ids of the quest giver and the quest taker. We made these optimizations and tested it on a 8 x c3.xlarge setup. We saw the error rate go even lower. Now we are able to support 2000 concurrent users without significant errors. Here are the results:



Also, now we the 2 sec latency point is hit at about 900 users as opposed to the 700 users earlier without the indices. It is interesting to note that this is about the same performance as the 8 x c3.4xlarge setup without the SQL optimizations.

5. Summary

Questmaster is a simple MVC application we built as part of this course. Even though the actual application was fairly simple, the scalability issues we faced were enlightening. Here are our lessons so far:

- Scaling up with a single server is feasible only up to a certain extent
- Scaling out i.e. adding more servers gives more or less linear scaling
- The slope of this linear curve is usually far less than 1

- It takes a lot of optimization to get the slope as close to 1 as possible
- Simple database optimizations like adding indexes are the best first-optimizations that one can do to improve scalability
- Web-server optimizations are highly dependent on the kind of instances the app is deployed on. Obvious lesson learnt the hard way.
- Coming up with realistic critical paths and load-testing scenarios is harder than it seems
- The black-box nature of Rails and EC2 setup makes it easy to implement our app, but very difficult to debug the deployments