CS291A: Scalable Internet Services

gTrack: Track Prices of Games on Steam

Nazmus Saquib
Udit Paul
Alex Ermakov

Graduate Students

Department of Computer Science

University of California Santa Barbara

December 6, 2018

Contents

Abstract

gTrack is an organized game tracking website that allows users to access information related to different games available on one of the world's largest video game platform, Steam. Each game on Steam that is available on gTrack shows to a user information such as the genre of the game, backgrounds, emotes and cards associated with the game, as well as price history of the game. In addition to viewing information about a game, registered users of gTrack can also share their opinions about games by commenting and up-voting(or down-voting). Registered users also have the ability to communicate with other users on the website using the associated chatroom, gChat. On the other hand, unauthenticated users have access to the information about the games as well as viewing comments of other users. An administrator of the website reserves the right to add and delete any user/game. The purpose of this website is to enable users make a decision about purchasing/accessing a game from Steam by providing all the relevant information about the game in a succint manner.

Chapter 1

Introduction

Steam [1] is the world's most popular PC game distribution platform that has 67 million monthly active players. Since January 2016, Steam has experienced a whooping 27 million first time purchasers. According to steamspy, there are currently 26,363 games available on Steam. Each of these games fall under different categories and contains different features such as backgrounds, emotes and cards. Furthermore, the prices of these games also vary over the course of time. These information become important for a user wanting to purchase a game from Steam. However, such information are not always present in a clear and concise manner on the Steam website.

The idea behind gTrack is to have a dedicated website built using Ruby on Rails [2] meant to serve interested users who would like to access a game on Steam. on gTrack, unauthenticated users get to see a list of all available games on Steam. Such user also get to see the ratings associated with the games. Once authenticated, users get to check necessary information such as price, genre, backgrounds, emotes and cards of the games. Users are also offered a search feature using which games could be sorted based on different categories. Additionally, users are allowed to comment on a game, see comments of other users about a game, up vote or down vote a game and interact with other users using a chat room. The sole purpose of this website is to deliver as much information about a game as possible to a user to facilitate her decision process while accessing/buying a game from Steam.

As Steam has a huge user base, it is practical to assume that a website such as gTrack would also draw attention of many such users. As such, scalability of the website with increasing number of users becomes a key issue. In this report, we first present some of the features available in our website. We then proceed to present the findings of various load tests we conducted to determine possible bottlenecks in our website. We present the results of various load tests and discuss the results in details.

Chapter 2

Webapp Description

The features available on gTrack can be depicted using a flowchart presented in Figure 1. The features are discussed in details below.

Using bcrypt, we implemeted a secure user *sign up* process that lets an interested visitor of the website become a registered user. Uniqueness of e-mail identification is ensured to prevent different users signing up with the same e-mail address.

Any user is able to see a detailed list of games that are present in Steam. The steam ID of a game is also recorded for easy viewing of the game in Steam. Various information associated with the game, such as cards, backgrounds, emotes, and price histories can also be viewed. As it is very common to look for games by genres or companies, two pages have been dedicated to summarize these information.

A registered user is allowed to *comment* on games and *like/dislike* games. However, any user can view previous comments made by other users on a game and also the total number of likes/dislikes associated with a game.

Registered users can access the in website chat room, *gChat*, to communicate with other users.

Any user can *search* for games based on various criteria, such as name, price range, company name, number of backgrounds/cards/emotes, average card/emote price, and system requirements.

2.1 Data Model

As most modern applications tend to be highly data-driven, we made sure we had a sufficiently large dataset to work with. The entity-relationship diagram of our application has been presented in Figure 2.2. We recorded information for 15453 games. There are 100 simulated users/gamers in our application, one being the admin. Each gamer can like/dislike a game and also comment on games. There are 775511 comments in total. Each game had price history associated with it. During seeding we used 436322 entries of price. The total number of backgrounds, cards, and emotes is 26066, 79133, and 33157. Apart from these, considerable amount of space was needed by the relations between games and genres/companies. The system requirements for each game was recorded too. The database contained ten types of processors, ten types of memories, and ten types of graphics cards. A few of the data

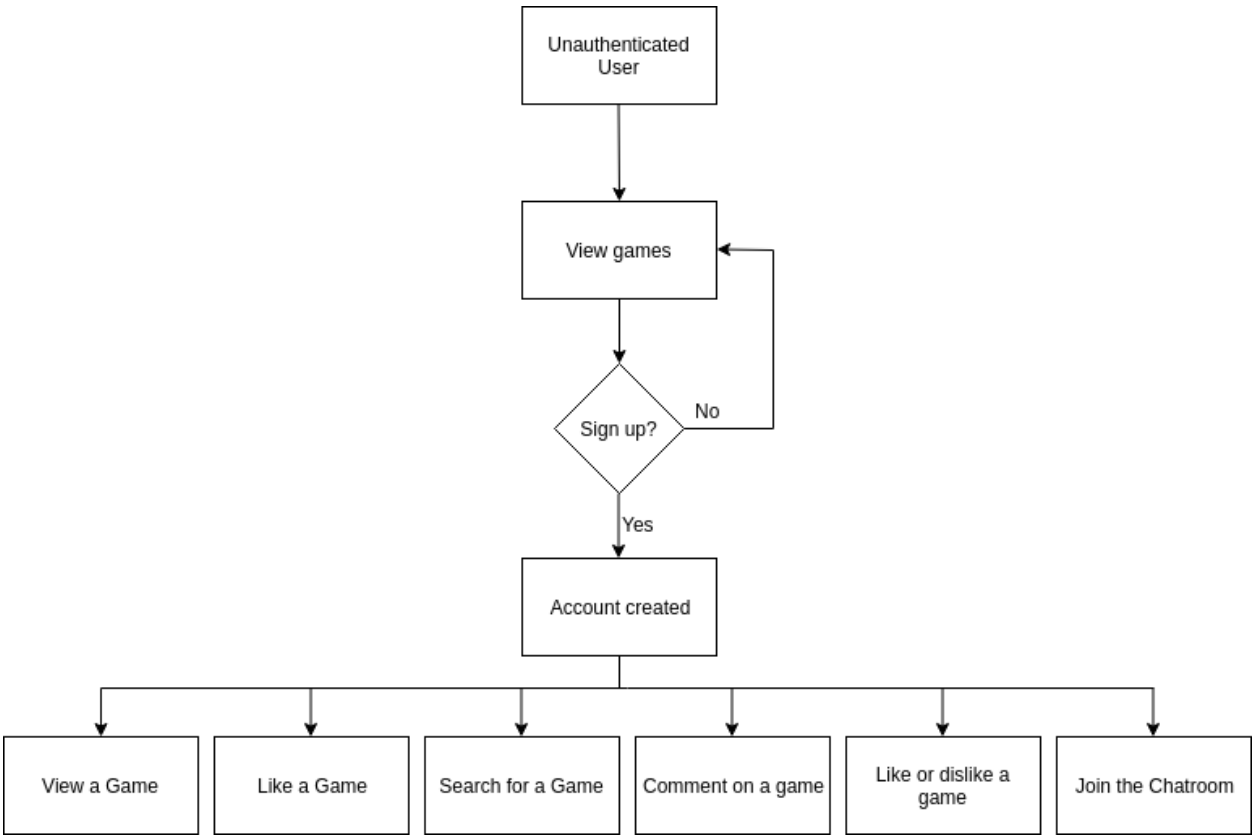


Figure 2.1: gTrack Flowchart

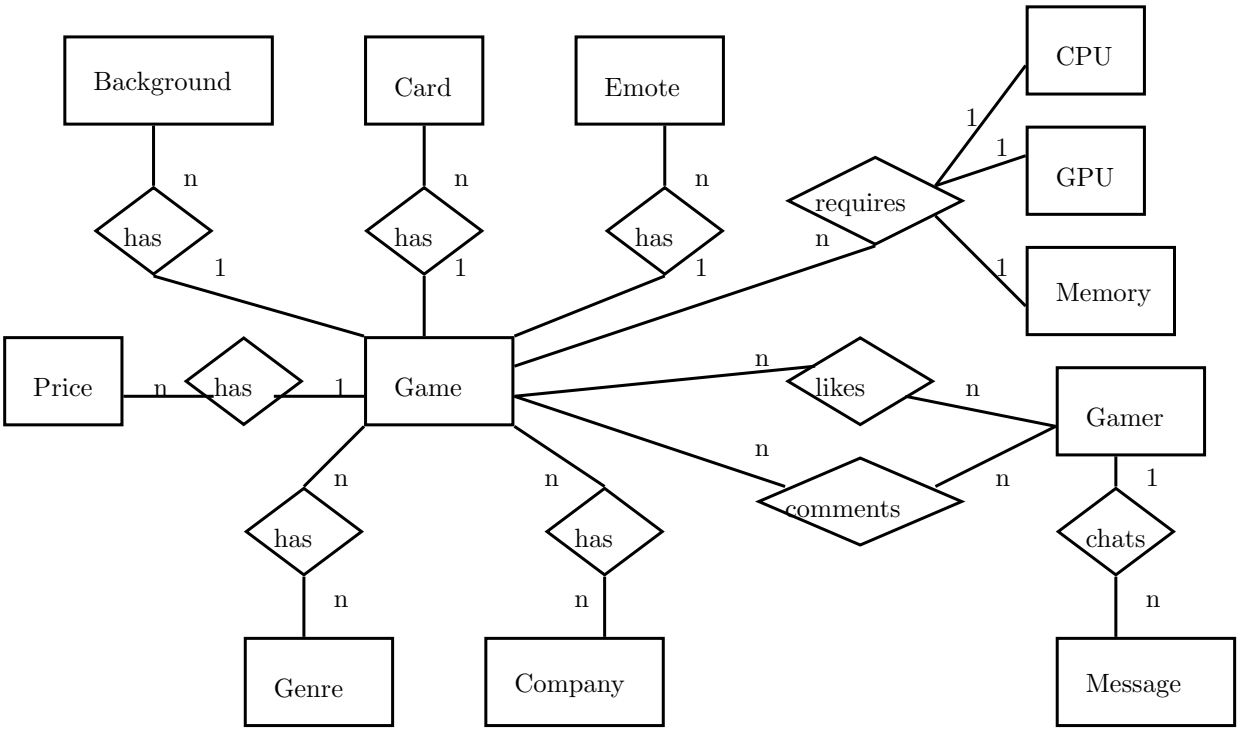


Figure 2.2: Entity-relationship diagram.

fields in the seed file were synthetic, however, most of the data were collected through steam API to make the model resemble real data as closely as possible. Due to the sheer amount of seed data, we had to create our own SQL scripts instead of relying on ActiveRecords of Rails as the latter turned out to be too slow. The seed file alone was 361MB, whereas the size of the database after seeding was 389MB.

Chapter 3

Load Testing

We used Tsung [3] framework for load testing. We designed a *generic workflow* which was able to capture the usual activities of a user while covering most, if not all, features of our website. We ran most of our tests on this generic workflow. Testing on the generic workflow gave us a rough idea on the overall performance of our website and possible bottlenecks. Later we created *specific workflow* separating out the portions that we thought might be causing the greatest bottlenecks. We ran our tests on this specific workflow before and after optimization to verify that this was indeed the workflow that was causing the bottlenecks and that these have been remediated by the optimizations. Unless otherwise specified, all the tests documented in this report were run on a C5 large app server and a m4 large database server. The optimizations described in this section are incremental, that is, stacked over the previous optimization.

3.1 Generic Workflow

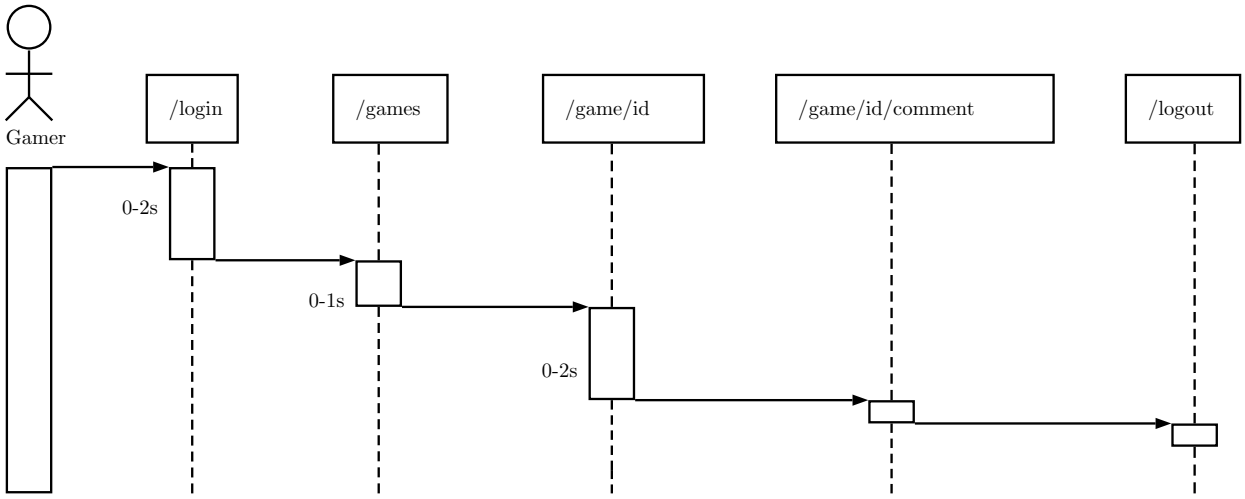
In this workflow, at first a user logs in to the website. The user waits for a time between zero and two seconds, visits the games index page, and then waits again for a time between zero and one second. Then the user randomly visits a game page, waits for a time between zero and two seconds, comments on that game and then logs out. This concludes the first sessions of our workflow.

In the second session, a user browses to the search page, searches for games by name, waits for a time between zero and one second, visits a random games page, waits for a time between zero and two seconds, searches for games by number of backgrounds, waits for a time between zero and two seconds, and visits a random game page.

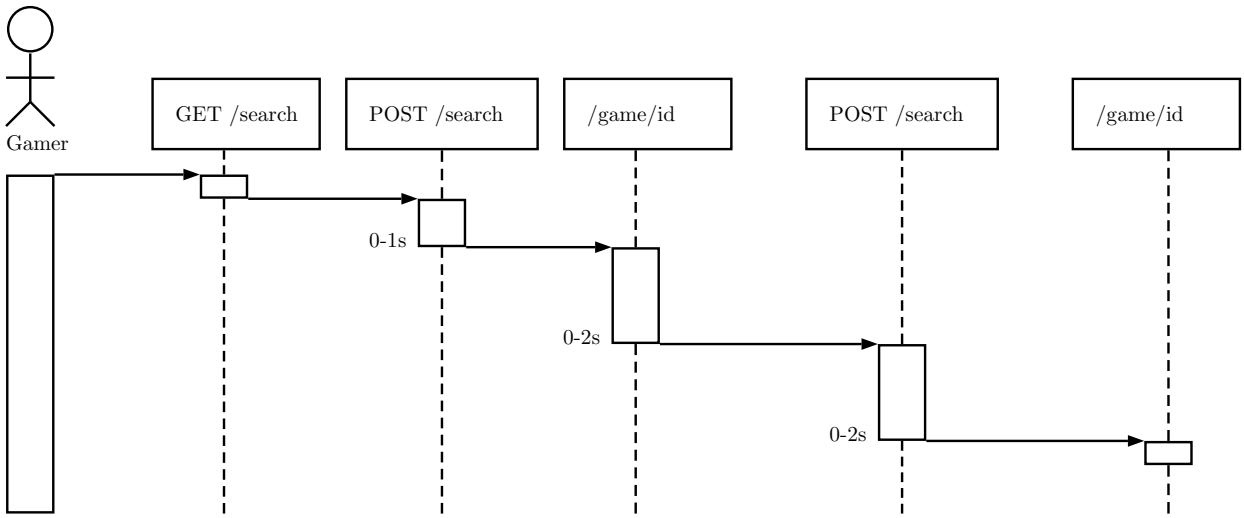
In the third session, a user browses to the gamers index page, waits between zero and one second, randomly gets a game, waits between zero and one second, browses to the genres page, waits between zero and one second, and finally browses to the companies page.

In the fourth session, a user browses to the signup page, waits for zero to three seconds, submits the form, waits for zero to one second, visits the games index page, waits for zero to one second, and finally likes/dislikes a random game.

For this workflow we had four arrival phases, the arrival rate being doubled at each consecutive phase and the very first one having an arrival rate of two users per second.



(a) First session.



(b) Second session.

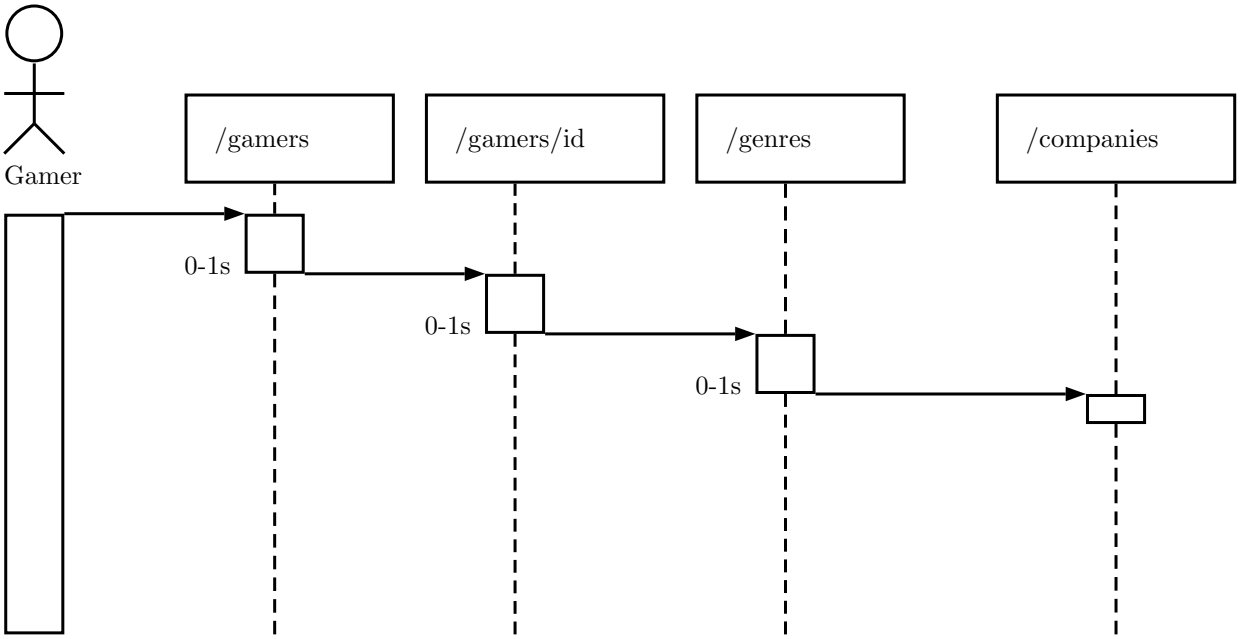
Figure 3.1: First and second sessions of generic workflow.

Figure 3.1 and 3.2 illustrate all the four sessions with the help of toned down sequence diagrams.

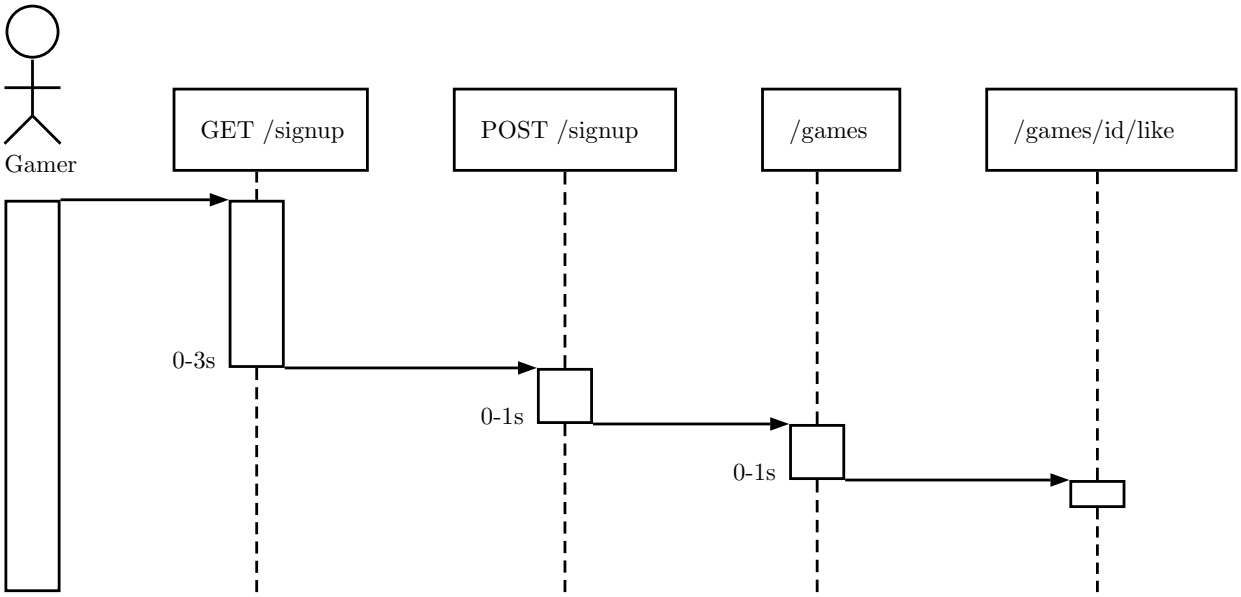
3.2 Specific Workflow

It came to our observation that the number of sql queries being invoked in a couple of pages (index page of genres and companies) was higher than normal. We hypothesized optimizing the query in some way might improve our response time. Just to verify our hunch, we came up with a specific workflow and tested that before and after optimization. The exact optimization has been discussed in a later section.

In this workflow, a user hits the index page for games, followed by the index pages of genres and companies. For this workflow we had four arrival phases, the arrival rate



(a) Third session.



(b) Fourth session.

Figure 3.2: Third and fourth sessions of generic workflow.

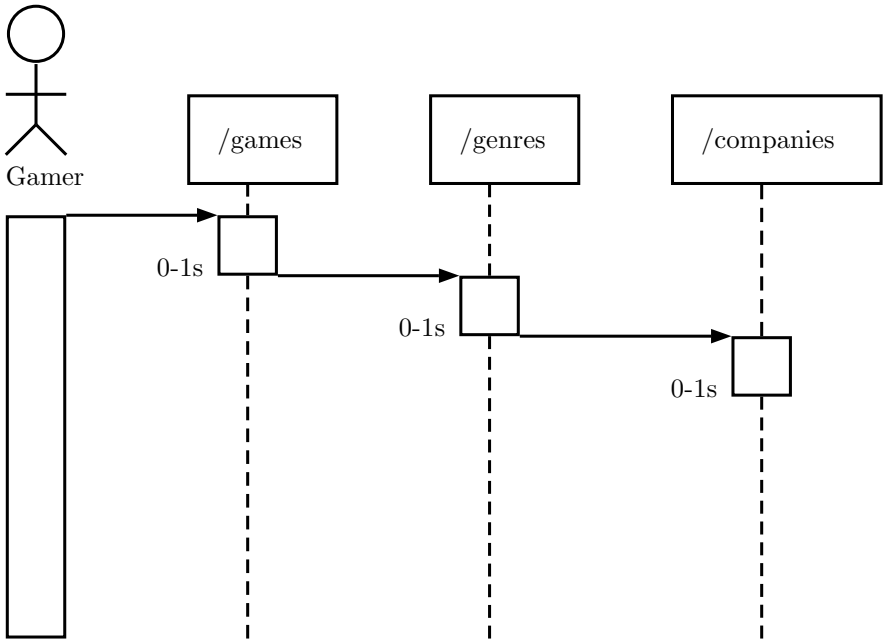


Figure 3.3: Specific workflow.

being doubled at each consecutive phase and the very first one having an arrival rate of two users per second. Figure 3.3 illustrates the specific workflow with the help of a toned down sequence diagram.

3.3 Optimization: AJAX

We observed a huge number of 5xx response codes the first time our application was tested using the generic workflow. A quick look at the rails log revealed that for every game showed in the index page, a query was executed to get its likes/dislikes. Instead of showing this information automatically, we decided to invoke an AJAX request on hover over "stats" text. It was observed that this increased the number of 200 codes and decreaed the number of 5xx codes. Figure 3.4 shows the total number of 200 and 5xx codes over time for deployment without and with AJAX.

3.4 Optimization: Pagination

Although the introduction of AJAX improved performance to some extent, the number of 5xx was still high. As the next step to optimization, we implemented pagination on the index page of games and comments page of individual games. Later on we implemented pagination on a few search pages too. Pagination increased the number of 200 codes further along with decreasing the number of 5xx codes. Figure 3.5 presents a graph of number of response codes along with a graph of mean duration of requests. It is evident that the introduction of pagination decreased the mean duration of requests. It is worth noting that in case of

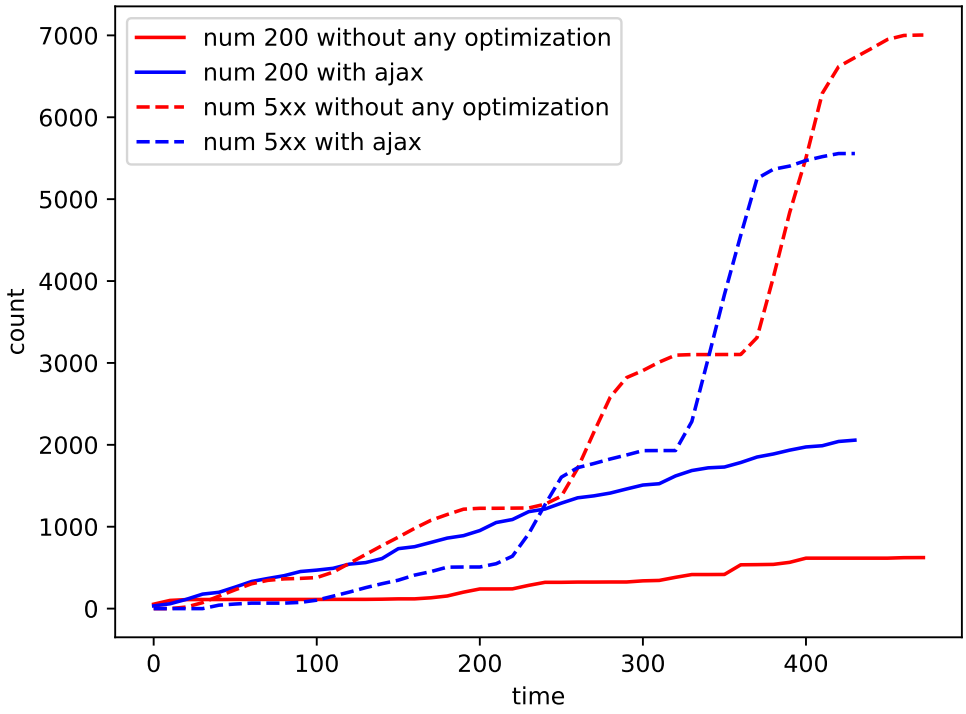


Figure 3.4: HTTP response counts with and without ajax.

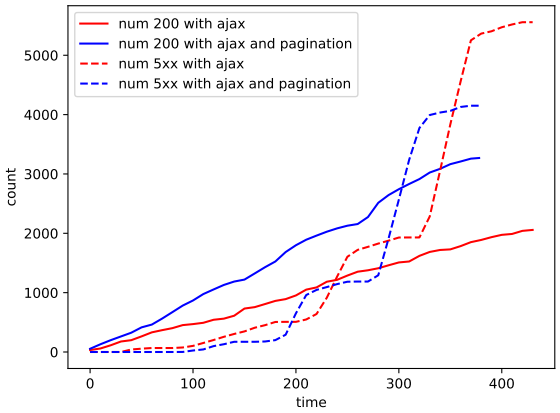
pagination the requests are serviced much faster, as a result of which it seems like there is a phase difference in the second graph and it completes much earlier.

3.5 Optimization: Database Indexing

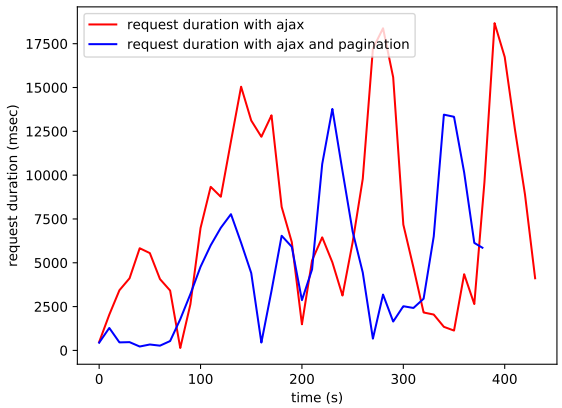
The data model of our application was complex. There were multiple different types of relations throughout the database. These relations were queried in one way or another in almost every page of the web app. Therefore we decided to index on the foreign keys of most tables. This resulted in a sharp boost in performance. Figure 3.6 shows the mean duration for search transaction (second session of generic workflow as discussed in Section 3.1) without and with indexing. It can be observed that indexing made the transaction significantly faster.

3.6 Optimization: SQL Query

Careful observation of the Rails log during development revealed that the "N + 1" problem of SQL query was prevalent in our application. Specifically, the genres and companies index pages shows the number of games under each genre/company. To retrieve this number, N more queries were triggered in the initial version of the app for N genres/companies apart from the one invoked to get all genres/companies. This situation was later remediated so that instead of "N + 1" queries only two queries were invoked. Tsung tests were run on the



(a) HTTP codes.



(b) Mean duration of requests.

Figure 3.5: Metrics for ajax only and ajax along with pagination.

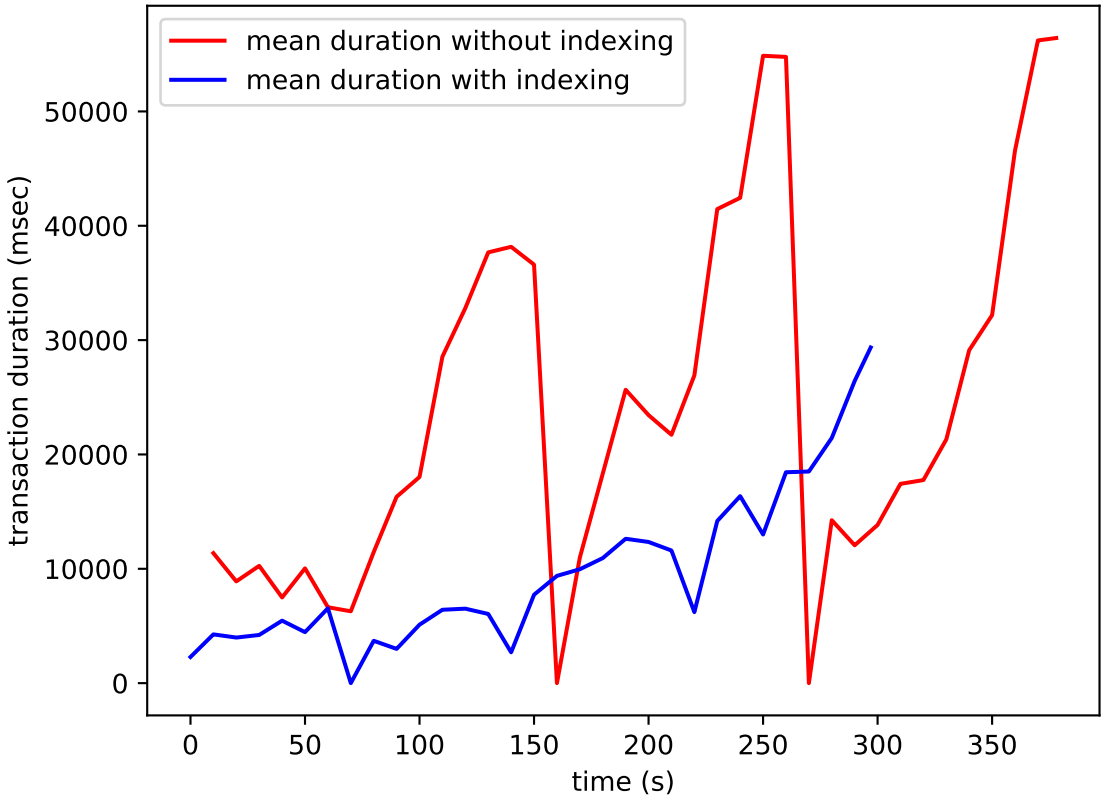


Figure 3.6: Mean duration for search transaction without and with indexing.

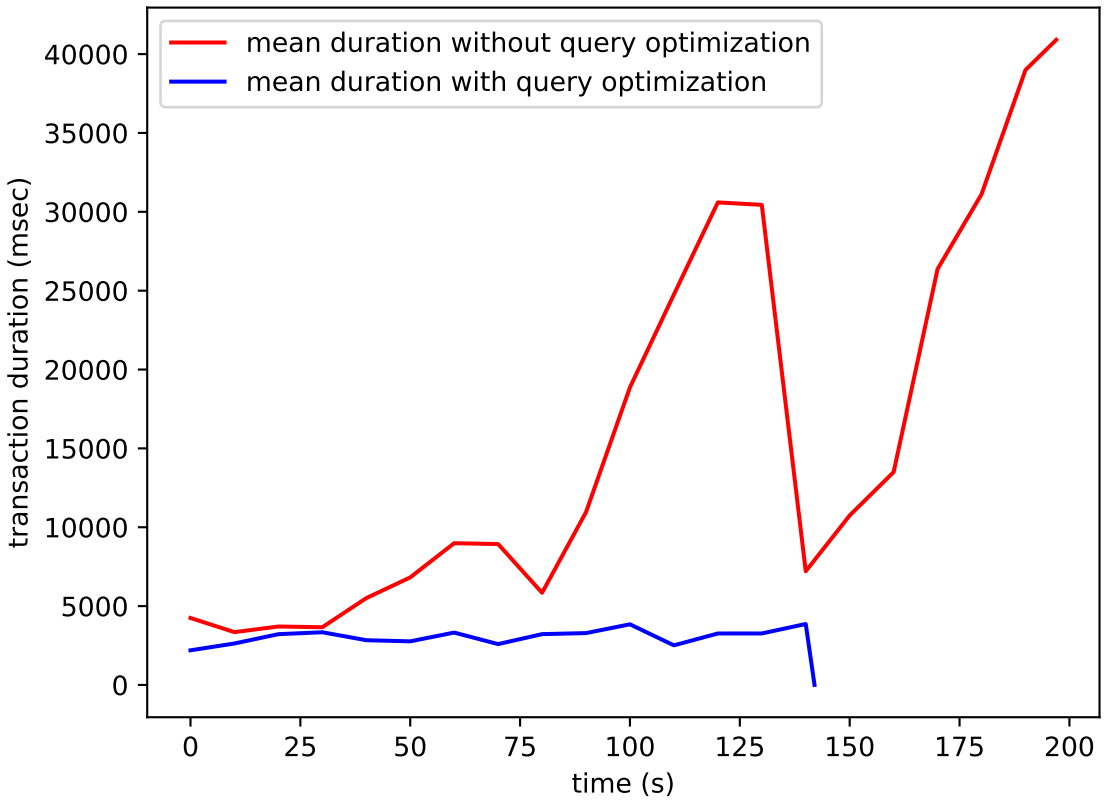


Figure 3.7: Mean duration for index page transaction without and with indexing.

specific workflow to evaluate the improvement in performance. Figure 3.13 shows the graph of mean duration for the transaction without and with indexing. It is evident from the graph that the discussed SQL optimization greatly improved the response time of the application.

3.7 Optimization: Caching

The gTrack web app has an intensive search feature. It was observed that one search – where a gamer provides his/her system requirements to get a list of games he/she can play, was taking particularly long time to respond. This is where we decided to apply Rails low level caching to speed up the query response. On a relevant note, as some of the searches could return a very large number of rows, the view had to be paginated. Otherwise it was observed that time taken for page rendering dwarved that for querying the database and the effect of cahing was not prominent. We found that Rails does not have an easy way to paginate results of custom query, so we had to come up with our own AJAX based pagination. To test our improvements we introduced a new workflow where users repeatedly make requests for that particluar search. It was observed that the response time decreased by a great extent as

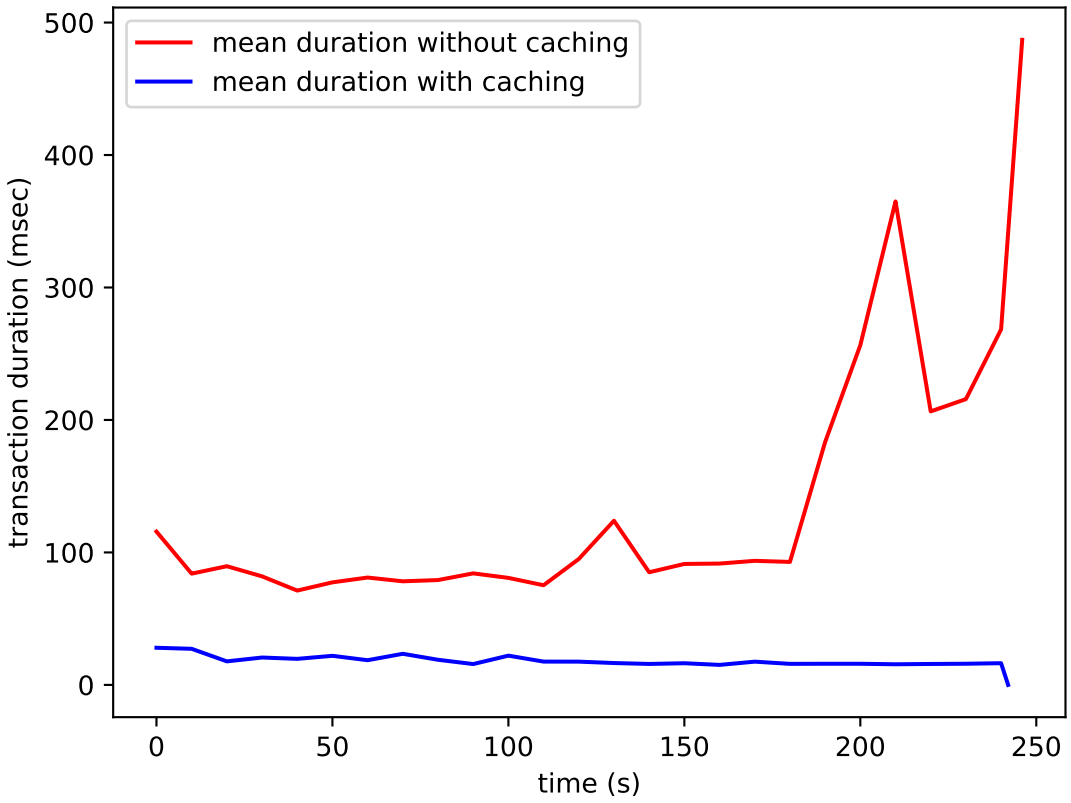


Figure 3.8: Mean duration for system requirement search transaction with and without caching.

a result of Rails low level query caching. Figure **??** shows the comparison between response time for the search page under discussion with and without caching. The corresponding Tsung file is presented in Appendix **??**.

3.8 Scaling

Once all the feasible optimizations were applied to our website, we ran Tsung load-test to obtain the ideal combination of hardware in AWS Elastic Beanstalk. The goal of this exercise was to determine the hardware (app server and database server) combination that was able to handle increasing number of users without producing server-related errors. We varied the number of a particular type of application server (c5 large) and the instance type of the database server. We ran the same Tsung XML file described in the Generic Workflow section on each of hardware configurations.

3.8.1 Results for up to 16 users/second arrival rate

In this test, the users arrival rate was modeled in four phases with phase 1 beginning with 2 users/second arrival rate. The arrival rate increased by a factor of 2 during subsequent

phases with phase 4 having 16 users arriving per second to carry out the transactions set out in the Tsung XML file.

We started off with a c5 large app server instance and a m4 large database server instance for this test. This combination yielded a significant number of 5xx errors at the end of the test (2612). In the next run, we increased the number of app server to 3 while keeping the same database server. This combination did not result in any significant improvement as we still had 2421 5xx errors. We then ran the test with 10 app servers (maintaining the same database instance type) and interestingly enough noticed a spike in the number of 5xx (2610). This result made it apparent that the true bottleneck of our website is in the database. To solve the problem, we tried a combination of 2 instances of a c5 large app server and m4.4xlarge database instance type. This was the hardware mixture that produced no 5xx error. This test also highlighted the delicate balance that exists between horizontal and vertical scaling. Figure 3.8 shows the mean response time that was achieved by different hardware configurations we tried during this test. From this figure,w e can see that apart from the ideal hardware combination, the other combinations took longer time on average to complete a request made by a user once the arrival rate increased.

Figure 3.9: Mean response time while handling up to 16 users/second

3.8.2 Results for up to 32 users/second arrival rate

The users' arrival rate was increased up to 32 per second for this test. We started by keeping the exact hardware combination that produced the best result while handling arrival rate of up to 16 users/second (2 instances of c5.large and a m4.4xl database instance). This combination resulted in 201 5xx errors. Before getting a bigger database instance, we tried to horizontally scale the app server to 4 instances of c5.large. The combination of 4 c5.large

app server instances and a m4.4xlarge database server was able to complete the test without resulting in any 5xx errors. Horizontal scaling worked perfectly for this test to be able to handle 32 users/second. Figure 3.9 shows the mean response time that was achieved by the two different hardware configurations we tried during this test. It can be observed from this figure that the horizontal scaling resulted in pages being rendered to users much quicker than the combination that produced some number of 5xx errors.

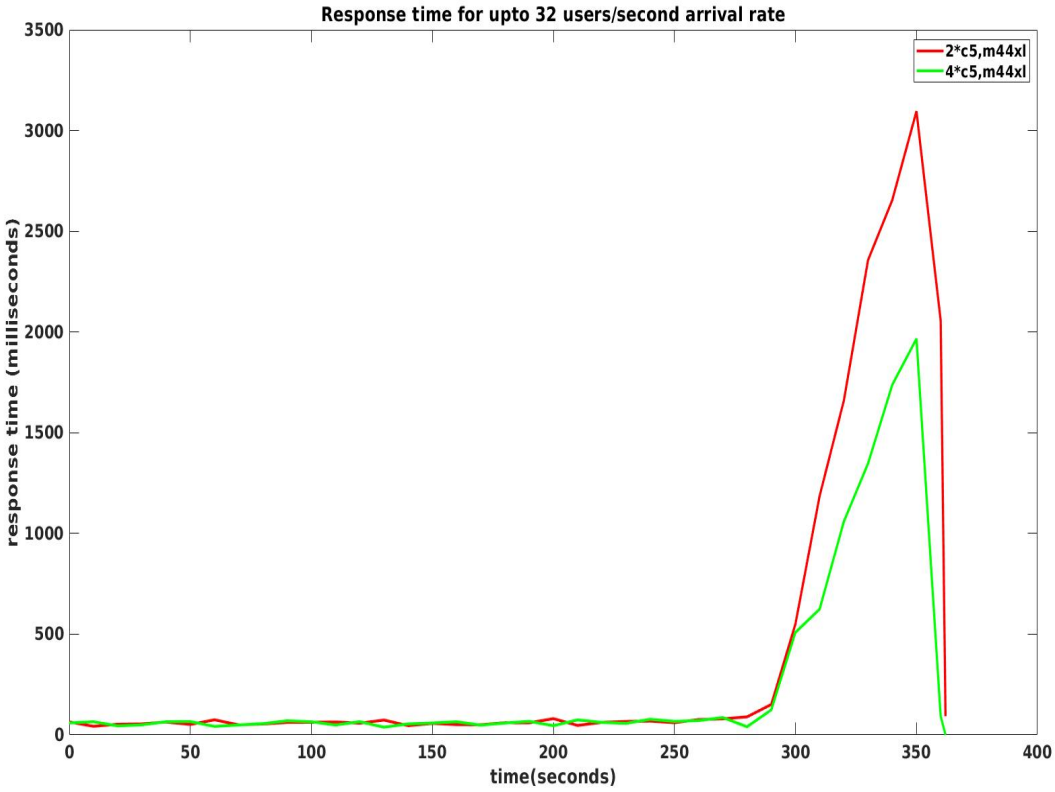


Figure 3.10: Mean response time while handling up to 32 users/second

3.8.3 Results for up to 64 users/second arrival rate

We increased the arrival rate by two fold from the previous test to test our website. This test contained 4 arrival phases with the last one having user arrival rate of 64 users/second.

The test was started with the hardware combination of 4 instances of c5.large application server with a m4.4xlarge database server. This combination produced a total of 5500 5xx error. We followed this by first increasing the number of app servers to 6 and then 8. As the 5xx errors remained, we used a different type of database server instance (m4.10xlarge) with 8 instances of c5.large app server instances. This setting produced no 5xx errors while being able to handle up to 64 users/second. Figure 3.10 demonstrates the mean response time obtained for each combination of hardware we applied during this test. Just as in previous test cases, we can notice that the combination that yielded with no 5xx error was

able to meet user generated requests much quicker as opposed to other combinations of hardware. An interesting trend is observed when we horizontally scaled the app server from 6 instances of c5.large to 8 instances. Even though we noticed a decrease in 5xx errors from the former, the latter actually took, on average, longer time to fulfill user requests. This potentially demonstrates that, aside from increased cost, adding more instances does not always produce desired result.

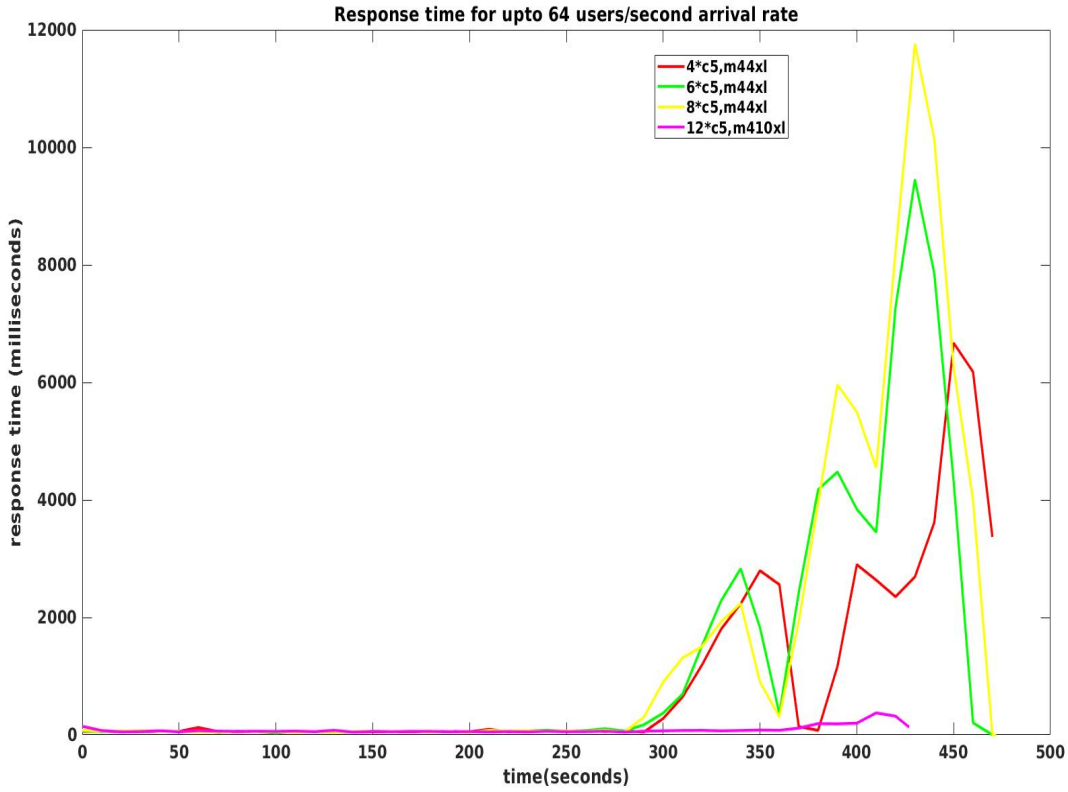


Figure 3.11: Mean response time while handling up to 64 users/second

3.8.4 Results for up to 128 users/second arrival rate

The arrival rate was increased to 128 users/second for this test. The first hardware combination used were 8 instances of c5.large app server and m4.10xlarge database server instance. This combination failed to handle the increased number of arrival rate. Subsequently,we tried 10 and 12 instances of c5.large app servers with m4.16xlarge database server instance. The ideal combination for this test that resulted in no 5xx errors were 12 instances of c5.large app server instances and m4.16xlarge of a database instance. Figure 3.11 shows the mean response time each of these combination was able to provide during this test. Here too we can observe that the best combination was able to meet user request in much better time than other combinations.

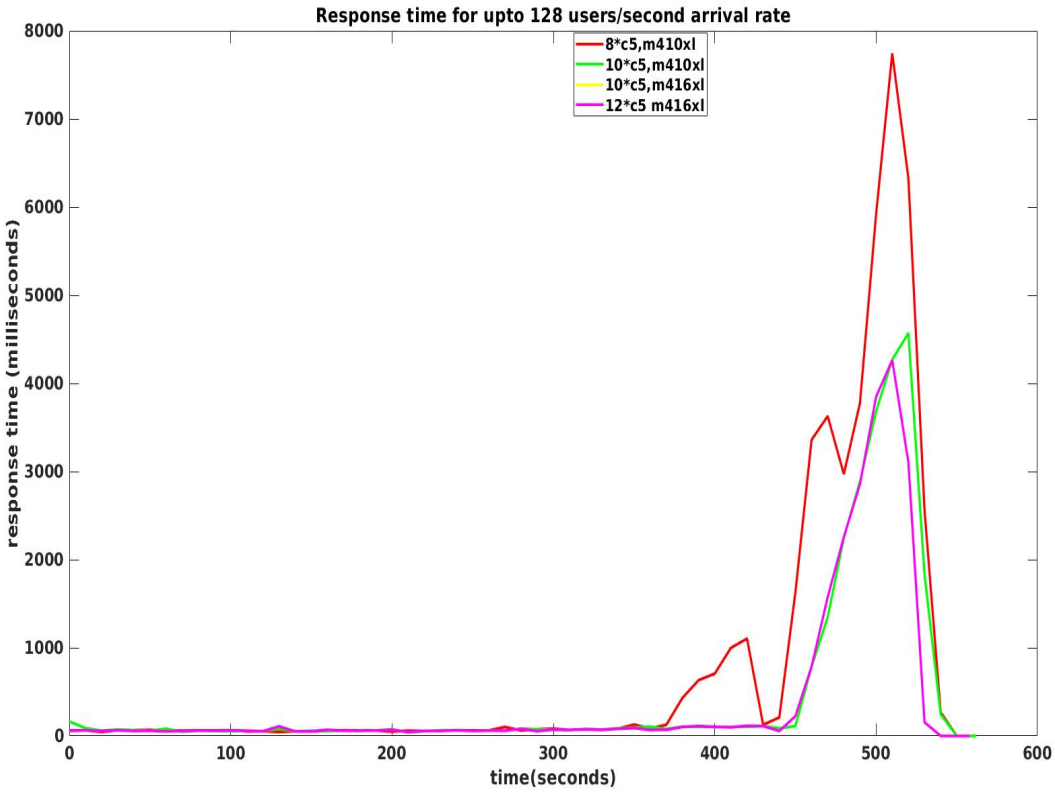


Figure 3.12: Mean response time while handling up to 128 users/second

3.8.5 Results for up to 256 users/second arrival rate

In this test we attempted to determine the hardware combination that could adequately handle 256 users/second arrival rate. In the beginning, We started with 12 instances of c5.large and a ,4.16xlarge app and data base server respectively. This fell well short of meeting the set arrival rate. To tried adding 2 additional app servers but that did not succeed as well. Finally we tried the m5.24xlarge database instance type with 14 instances of c5 large app server. This resulted in only 6% of 5xx errors. At this point, we attempted to utilize a different app server to see if we could overcome the barrier all together. However, our attempt to use the c5d.18xlarge app server did not succeed as the instance failed to properly load up. Figure 3.12 shows the mean response times that were noticed during the different hardware combinations we attempted for this test. From figure, we can conclude that to meet with 256 user requests per second, we need a hardware configuration closer to 14 instances of c5.large app server and a m5.24xlarge database server.

3.8.6 Cost Benefit Analysis

Figure 3.13 shows the combination of the hardware that yielded the best result for different number of users' arrival rates per second. This figure also demonstrates the daily cost

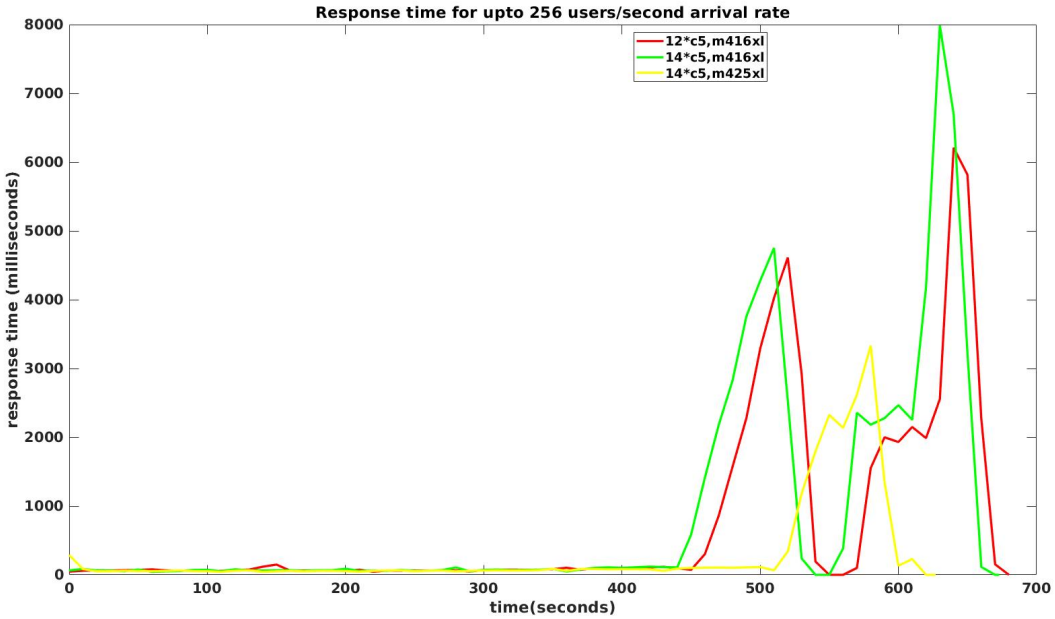


Figure 3.13: Mean response time while handling up to 256 users/second

associated with using each of this hardware combination, thereby easing the process of computing the cost of supporting a certain number of user at any second. Each of these hardware combinations was able to complete the Tsung load test without producing any 5xx HTTP error responses. For example, to meet up with 128 users per second (which could happen during peak time), the hardware combination of 12 c5.large app server instances with a m4.16xlarge database instance is seen to perform best. This combination cost about 7 USD per hour. On the other hand, we can see from Figure 3.13 that to meet up with 32 users/second, we only need to spend around 3 USD. Using tis information, we can be better prepared to handle various levels of users' arrivals during different times of the day.

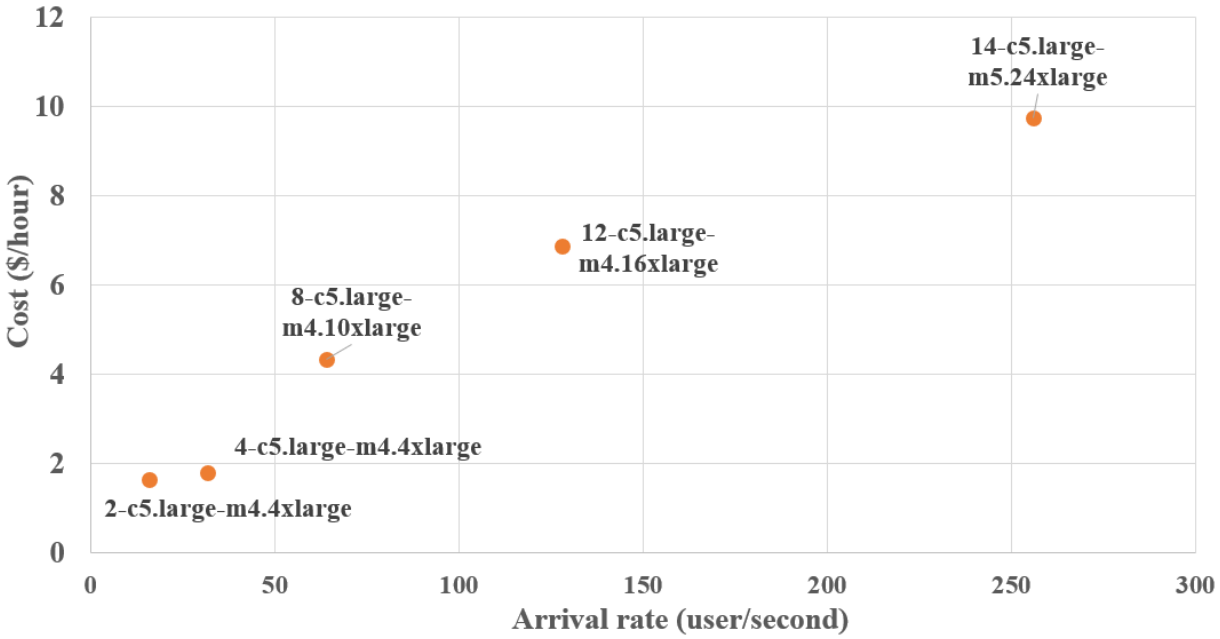


Figure 3.14: Cost against user arrival rate for different AWS hardware configurations

Chapter 4

Future Work

From a broader perspective, there are are two major categories of improvements that can be performed as future work: one category of improvement is related to feature of the app, another category is related to improving the scalability of the app.

In regards to the first category of improvement, one improvement we can make is to introduce a version of global chat which is workable in Elastic Beanstalk [4]. Although our current version of the global chat works correctly locally, we need to refresh the page after each new message in Elastic Beanstalk. Additionally, we could try to add more features throughout various portions of our web app. For example, right now the user profile is quite rudimentary. It does not have any user related info other than name, email, password, and possibly gravatar. We could add a *watchlist* for each user, which is basically a list of games that the user is tracking. This list of games along with the associated price histories would ideally pop up once the user browses to his/her profile page.

In regards to the second category of improvement, we can try out different sorts of caching. Although we were able to manually show the effects of one type of caching only, ideally we would want to show the effects of different types of caching with the help of an automated tool.

Chapter 5

Conclusion

Recent days have seen an upsurge in the number of data-driven applications. With the prominence of these applications, scalability has posed itself as a major concern. In this project, we tried to demonstrate how we can make our application scalable. We proposed some methods for enhancing the scalability of our application. We justified the employment of these methods with the help of load-testing our application using Tsung Framework.

While load-testing our application, we tried to construct generic scenarios which we deemed to be common actions for most users. We followed an iterative approach to improvement. That is, we first test our application and try to find a bottleneck. Once we localte a bottleneck, we try to remediate it. Once it has been taken care of, we run the tests again. This process continues as long as it is feasible to achieve better performance in the next iteration over the previous one.

In particular, we used the following techniques for improving the performance of our web app: pagination, AJAX, indexing, query optimization, horizontal and vertical scaling, and caching. The sheer amount of seed data that we had alone created obstacles in reasonable performance improvement. However, we believe our data model closely resembles most modern websites that tend to be heavily data-driven. Although we have tested our web application extensively with Tsung, we have to keep in mind that Tsung is not capable of simulating many aspects of a client interacting with a browser. Apart from that, we are fairly confident that we were able to introduce necessary improvements for boosting the performance of our web application and support it through Tsung testing.

Appendix A

Tsung Files

A.1 Tsung File for General Workflow

Listing A.1: Test

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/local/share/tsung/tsung-1.0.dtd" [] >
<tsung loglevel="notice">

  <!-- Client side setup -->
  <clients>
    <client host="localhost" use_controller_vm="true" maxusers='15000'/>
  </clients>

  <!-- Server side setup -->
  <servers>
<server host="tsap.zm7nup4fs2.us-west-2.elasticbeanstalk.com" port="80"
    type="tcp"></server>
  </servers>


  <load>
    <arrivalphase phase="1" duration="1" unit="minute"
        wait_all_sessions_end="true">
      <users arrivalrate="2" unit="second"></users>
    </arrivalphase>

    <arrivalphase phase="2" duration="1" unit="minute"
        wait_all_sessions_end="true">
      <users arrivalrate="4" unit="second"></users>
    </arrivalphase>

    <arrivalphase phase="3" duration="1" unit="minute"
        wait_all_sessions_end="true">
      <users arrivalrate="8" unit="second"></users>
    </arrivalphase>

    <arrivalphase phase="4" duration="1" unit="minute"
        wait_all_sessions_end="true">
```

```
30        <users arrivalrate="16" unit="second"></users>
31      </arrivalphase>
32  <!--
33      <arrivalphase phase="5" duration="1" unit="minute">
34        <users arrivalrate="32" unit="second"></users>
35      </arrivalphase>
36
37      <arrivalphase phase="6" duration="1" unit="minute">
38        <users arrivalrate="64" unit="second"></users>
39      </arrivalphase>
40  -->
41    </load>
42
43    <options>
44      <!-- Set connection timeout to 2 seconds -->
45      <option name="file_server" id="gamesFile" value="games.csv"/>
46      <option name="file_server" id="gamersFile" value="gamers.csv"/>
47      <option name="file_server" id="truefalseFile" value="truefalse.csv"/>
48      <option name="global_ack_timeout" value="2000"></option>
49      <option type="ts_http" name="user_agent">
50        <user_agent probability="100">Mozilla/5.0 (Windows; U; Windows NT
              5.2; fr-FR; rv:1.7.8) Gecko/20050511 Firefox/1.0.4</user_agent>
51      </option>
52    </options>
53
54    <sessions>
55    <!--ONE-->
56      <session name="http-example" probability="20" type="ts_http">
57        <setdynvars sourcetype="file" fileid="gamesFile" delimiter="," order
              ="random">
58          <var name="gameid" />
59          <var name="appid" />
60        </setdynvars>
61        <setdynvars sourcetype="file" fileid="gamersFile" delimiter=","
              order="random">
62          <var name="user" />
63          <var name="pass" />
64        </setdynvars>
65          <transaction name="post_comment_on_game">
66            <request subst="true">
67              <http url="/login" method="POST" version="1.1"
68                  contents="session[email]=%%_user%%&amp;
69                  session[password]=%%_pass%%&amp;
70                  commit=Log+in"
71                  content_type="application/x-www-form-urlencoded" />
72            </request>
73            <thinktime value="2" random="true"></thinktime>
74            <request>
75              <http url="/games" method="GET" version="1.1"/>
76            </request>
77            <thinktime value="1" random="true"></thinktime>
78            <request subst="true">
79              <http url="/games/%%_gameid%%" method="GET" version="1.1"/>
80            </request>
```

```
81          <thinktime value="2" random="true"></thinktime>
82          <request subst="true">
83            <http url="/games/%%_gameid%%/comments" method="POST" version=
                "1.1" contents="comment[description]=The+quick+brown+fox&
                amp;commit=Submit+Comment" />
84          </request>
85          <request>
86            <http url="/logout" method="DELETE" version="1.1" />
87          </request>
88
89        </transaction>
90    </session>
91  <!--TWO-->
92    <session name="http-example" probability="30" type="ts_http">
93      <setdynvars sourcetype="file" fileid="gamesFile" delimiter="," order
            ="random">
94        <var name="gameid" />
95        <var name="appid" />
96        </setdynvars>
97        <transaction name="searchGames">
98        <request subst="true">
99            <http url="/search_pages/search" method="GET" version="1.1" />
100         </request>
101         <request>
102           <http url="/search_pages/search" method="POST" version="1.1"
103               contents="name=Counter&amp;
104               search_type=game_by_name" />
105         </request>
106         <thinktime value="1" random="true"></thinktime>
107         <request subst="true">
108           <http url="/games/%%_gameid%%" method="GET" version="1.1" />
109         </request>
110         <thinktime value="2" random="true"></thinktime>
111         <request>
112           <http url="/search_pages/search" method="POST" version="1.1"
113               contents="maximum_Background=2&amp;
114           minimum_Background=1&amp;
115           search_type=game_by_background_count" />
116         </request>
117         <thinktime value="2" random="true"></thinktime>
118         <request subst="true">
119           <http url="/games/%%_gameid%%" method="GET" version="1.1" />
120         </request>
121       </transaction>
122   </session>
123 <!--THREE-->
124   <session name="http-example" probability="48" type="ts_http">
125     <setdynvars sourcetype="random_number" start="1" end="100">
126       <var name="rndint" />
127     </setdynvars>
128       <transaction name="getListPages">
129         <request>
130           <http url="/gamers" method="GET" version="1.1" />
131         </request>
```

```
132           <thinktime value="1" random="true"></thinktime>
133           <request subst="true">
134             <http url="/gamers/%%_rndint%%" method="GET" version="1.1" />
135           </request>
136           <thinktime value="1" random="true"></thinktime>
137           <request>
138             <http url="/genres" method="GET" version="1.1" />
139           </request>
140           <thinktime value="1" random="true"></thinktime>
141           <request>
142             <http url="/companies" method="GET" version="1.1" />
143           </request>
144         </transaction>
145     </session>
146   <!--FOUR-->
147     <session name="http-example" probability="2" type="ts_http">
148       <setdynvars sourcetype="random_string" length="10">
149         <var name="rndname" />
150         <var name="rndemail" />
151       </setdynvars>
152       <setdynvars sourcetype="file" fileid="gamesFile" delimiter="," order
              ="random">
153         <var name="gameid" />
154         <var name="appid" />
155       </setdynvars>
156       <setdynvars sourcetype="file" fileid="truefalseFile" delimiter=","
              order="random">
157         <var name="tfvar" />
158       </setdynvars>
159         <transaction name="createAccount">
160           <request subst="true">
161             <http url="/signup" method="GET" version="1.1" />
162           </request>
163           <thinktime value="3" random="true"></thinktime>
164           <request subst="true">
165             <http url="/gamers" method="POST" version="1.1"
166               contents="commit=Create+my+account&amp;
167                         gamer[email]=%%_rndemail%%@gamer.com&amp;
168                         gamer[gamername]=%%_rndname%%&amp;
169                         gamer[password_confirmation]=123456&amp;
170                         gamer[password]=123456" />
171           </request>
172           <thinktime value="1" random="true"></thinktime>
173           <request>
174             <http url="/games" method="GET" version="1.1" />
175           </request>
176           <thinktime value="1" random="true"></thinktime>
177           <request subst="true">
178             <http url="/games/%%_gameid%%/like" method="POST" version="1.1
                    " contents="like=%%_tfvar%%" />
179           </request>
180
181         </transaction>
182     </session>
```

```
183   </sessions>
184 </tsung>
```

A.2 Tsung File for System Requirement Query

Listing A.2: Test

```
1 <?xml version="1.0"?>
2 <!DOCTYPE tsung SYSTEM "/usr/local/share/tsung/tsung-1.0.dtd" [] >
3 <tsung loglevel="notice">
4
5   <!-- Client side setup -->
6   <clients>
7     <client host="localhost" use_controller_vm="true" maxusers='15000'/>
8   </clients>
9
10  <!-- Server side setup -->
11  <servers>
12        <server host="tsnap.zm7nup4fs2.us-west-2.elasticbeanstalk.com"
              port="80" type="tcp"></server>
13  </servers>
14
15
16   <load>
17    <arrivalphase phase="1" duration="60" unit="second"
          wait_all_sessions_end="true">
18      <users arrivalrate="2" unit="second"></users>
19    </arrivalphase>
20
21    <arrivalphase phase="2" duration="60" unit="second"
          wait_all_sessions_end="true">
22     <users arrivalrate="4" unit="second"></users>
23    </arrivalphase>
24
25    <arrivalphase phase="3" duration="60" unit="second"
          wait_all_sessions_end="true">
26      <users arrivalrate="8" unit="second"></users>
27    </arrivalphase>
28
29    <arrivalphase phase="4" duration="60" unit="second"
          wait_all_sessions_end="true">
30      <users arrivalrate="16" unit="second"></users>
31    </arrivalphase>
32
33 </load>
34
35  <options>
36    <option name="file_server" id="gamesFile" value="games.csv"/>
37    <option name="global_ack_timeout" value="2000"></option>
38    <option type="ts_http" name="user_agent">
39      <user_agent probability="100">Mozilla/5.0 (Windows; U; Windows NT
            5.2; fr-FR; rv:1.7.8) Gecko/20050511
```

```
40        Firefox/1.0.4
41       </user_agent>
42     </option>
43   </options>
44
45   <sessions>
46     <session name="http-example" probability="100" type="ts_http">
47          <request subst="true">
48             <http url="/search_pages/search" method="POST" version="1.1"
49                  contents="graphic=1&amp;
50                            memory=1&amp;
51                            processor=1&amp;
52              search_type=sys_req"/>
53           </request>
54     </session>
55
56   </sessions>
57 </tsung>
```

Bibliography

[1] Welcome to Steam. https://store.steampowered.com/. [Online; last accessed 03-December-2018].

[2] Ruby on Rails. https://rubyonrails.org/. [Online; last accessed 03-December-2018].

[3] Tsung. http://tsung.erlang-projects.org/. [Online; last accessed 03-December-2018].

[4] AWS ElasticBeanstalk – Deploy Web Applications. aws.amazon.com/AWS/Beanstalk. [Online; last accessed 03-December-2018].