LoLCounter

Statistical Data Analysis Site for League of Legends matches

CS188 Scalable Internet Services

Prof. Andrew Mutz

Fall 2015

Team

Paul Kim

Alex Wang

Daniel Yao

Roland Zeng

Repository:

https://github.com/scalableinternetservices/lolcounter

Agile Tracker:

https://www.pivotaltracker.com/n/projects/1446700

1 Introduction

LoLCounter is a web service that analyzes match data from League of Legends. League of Legends is a multiplayer online battle arena (MOBA) created by Riot. Each game consists of two teams of five people, battling each other in an attempt to destroy the opposing side's base. Each person selects one of the 128 available champions, and picks a role/lane - Top, Jungle, Mid, ADC, or Support. Given this setting, and the fact that every one of the millions of matches are stored on Riot's API, we wished to analyze as many games as we could to establish meaningful patterns of data. For example, when Teemo, a champion, enters Top lane, which opposing top champion wins the most against him? When a certain combination of 5 champions plays on the same team, what is the probability they will win?

The core of LoLCounter is one single database of matches. These matches are pulled from Riot's API and stored in our database. Currently, there are 300,000+ matches stored. Each database entry contains crucial game data, such as which champions were involved and which team won. Then, using various database queries dependent on user input, we can return relevant data such as individual champion win-rates, champion counter rates, and team composition winrates. These features are explained in further detail below.

2 Description

The home page simply notifies the user of the current League of Legends scene. It provides information pertaining to new champions, the current free champions, and the most recent game updates/patches. Clicking on a new champion portrait opens a modal that includes the champion's abilities and its champion spotlight (an intro video created by Riot).

The summoner page allows the user to search for their account name. Whenever a user searches for a summoner, we gather the user's ranking, as well as information of their three most played champions from Riot's database using their API.

The champions page is a way for users to see a snapshot of how any champion has been performing. From this page, they can see a champion's win rate on any lane. For example, one may see that the champion Annie has won 53.3% of the 3535 games played on Mid lane.

The counterpicks page lets a user search for a champion and a specific lane. It then returns the top three highest-winrate champions against the searched champion. We imagine that if a user is in champion select (the lobby prior to entering a game, where users select their champion), and he sees that the enemy has selected Olaf and is most likely going top, he can search for Olaf Top in the counterpicks page to see what champions do well against him.

Team comp works similarly to counterpicks but on a larger scale. Instead of seeing how a champion competes in relation to other champions, we can show how well an entire team composition performances overall.

The summoner page makes remote API calls to Riot's database. The champions, counterpicks, and team comp page utilizes pre-processed data from LoLCounter's matches database.

3 Development

To create LoLCounter, we attempted to follow an agile development methodology. We worked in one-week sprints and held backlog grooming sessions every Tuesday for eight weeks. At first, we tracked tasks and assigned responsibility using a shared Google Sheets spreadsheet file. Then, after we received access to Pivotal Tracker, we began using it to write formal user stories. However, because we were not serving any customers or stakeholders, we would often add small aesthetic or performance features without writing a corresponding user story. Furthermore, because Pivotal Tracker was used mainly for our benefit instead of our (nonexistent) customers, our stories were often casually written instead of following the formal agile template.

4 Scalability issues

We faced many scalability issues with LoLCounter. Although most of our database calls relied on a single database, this database was huge. There are 128 champions that could go in 5 possible lanes, leading to hundreds of thousands of different combinations of win rates and millions of possible team compositions. This led to massive page load times for our champions page, which had to display every champion's win rate in every lane. Every time we attempted to load the champions page, we had to wait for more than 20 seconds. This is the biggest issue we attempted to optimize over the duration of this course.

5 Tsung Tests and Critical User Path

To ensure that our tsung test results reflected the benefits of our optimizations properly, we had to create a critical user that could be run between each optimizations. The critical user also had to follow a path that was reasonable but also displayed our optimizations properly. More specifically, all of our tsung tests were run on a m1.medium instance with the app also deployed on an m1.medium instance(except for the vertical scaling test). The tsung test consisted of four phases,each running for 30 seconds. Each phase increased the number of users per second by one, with the first phase starting at one. Each user accessed every part of the app five times, with at most a 2 second break between each get requests. We had the user revisit pages multiple times to highlight the power of client side caching but this caused users to have a relatively long session duration(which is a more realistic user experience than arriving and leaving in a short span of time for our app).

6 Pre-Optimization Tsung Testing

To properly test our app's scalability, we needed a baseline speed. To create this, we ran our tsung test on an app version without any of our optimizations except database optimization (reasoning discussed in section 7).

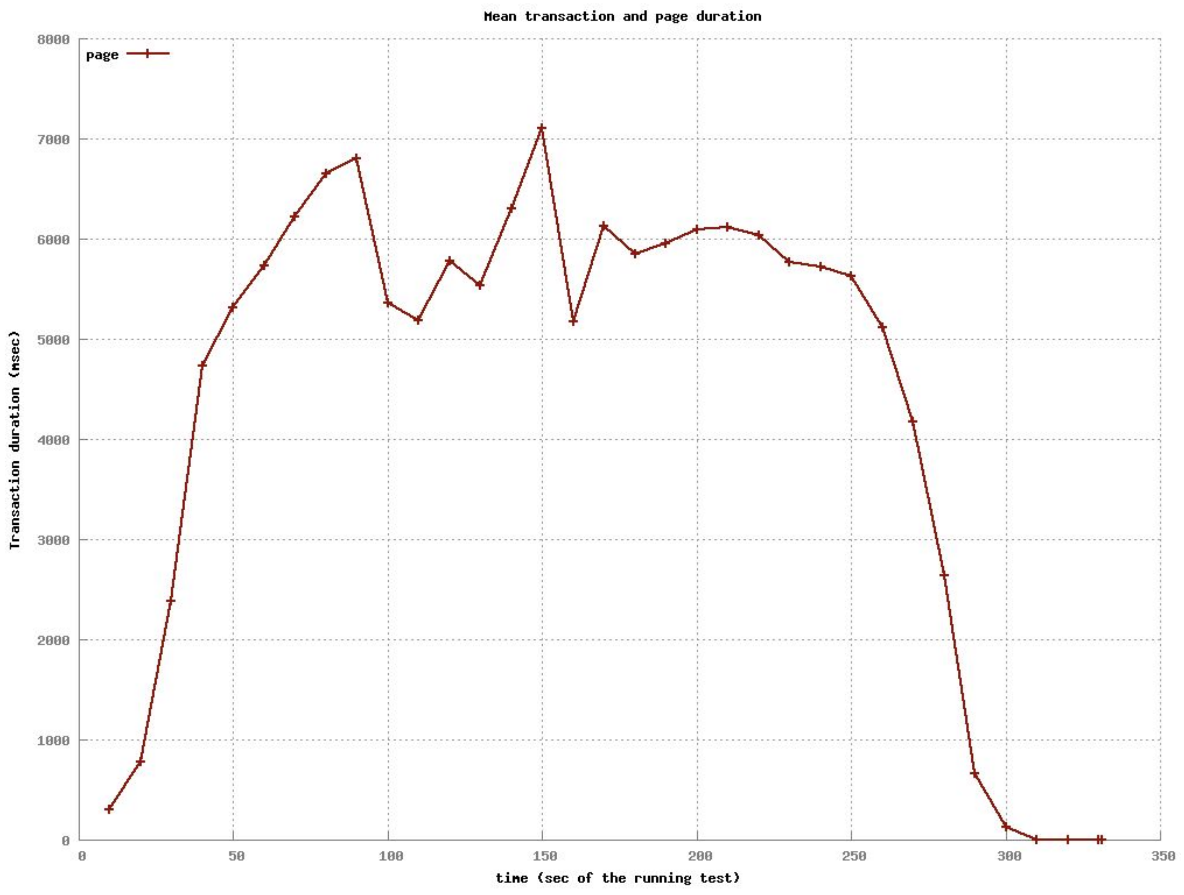
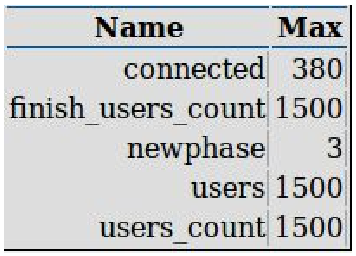


Figure 6.1: Mean transaction and page duration of tsung test on the application with no optimizations

From the graph, we can see that the response times max out around seven seconds with a mean response time of 5.07 seconds. Studies show that users will most likely leave a site after four seconds of waiting. Clearly, our response rates were abysmal at best and is too slow for any respectable web application. Although the response times were slow, all users successfully finished. This is most likely because our app only requires get requests which is very unlikely to cause any type of error. To further investigate our success rates, we ran a different tsung test with a much greater user mass. Even with 1500 users within a span of 40 seconds, all users successfully finished(Although some took up to 35 minutes).



Name	Max
connected	380
finish_users_count	1500
newphase	3
users	1500
users_count	1500

Figure 6.2: User data of tsung test with high user number on an unoptimized application

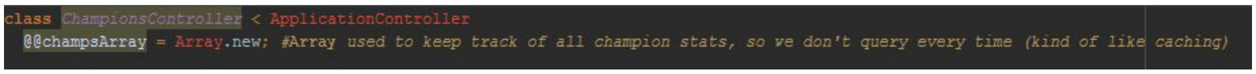
7 Optimization: database structure

Our initial database consisted of only a games table. When a user requested a champion's counters, we looked at all games and calculated the win percentage of all other champions against that champion. For example, to find Annie's counters, we had to query for all games that Annie was involved and calculate which champions had the highest win rate against her. Because the database of games are mostly static, we instead created a static table consisting of counters information that we calculated beforehand. Now, we only need to calculate counters every time we update our games table rather than every request and searching for Annie's counters consists of a single query on the counters table rather than the entire games table.

Our database had too many games that running a tsung test without this database optimization was not viable. The tests could not function properly and resulted in very few users succeeding. Because of this, all tsung tests including the pre-optimization tests used a database with the counters table.

8 Optimization: server side caching

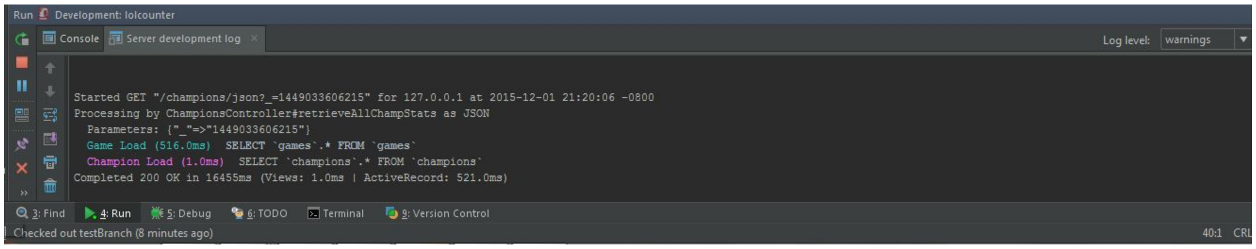
As discussed in section 4, we were encountering massive load times when we accessed the champions page. However, we realized that this data would rarely change, and so we could safely cache it on the server.



```
class ChampionsController < ApplicationController
  @@champsArray = Array.new; #Array used to keep track of all champion stats, so we don't query every time (kind of like caching)
```

Figure 8.1: /app/controllers/champions_controller.rb

Because the backend game's table is not updated often, we thought it to be safe to store the results used to generate the Champion's table within the server.The initial call to /champions endpoint used to generate the Champion Table takes 16,455ms.

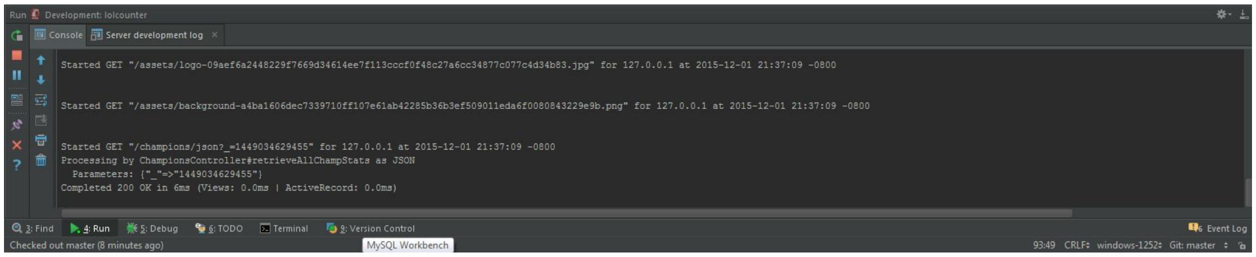


```
Started GET "/champions/json?_=1449033606215" for 127.0.0.1 at 2015-12-01 21:20:06 -0800
Processing by ChampionsController#retrieveAllChampStats as JSON
  Parameters: {"_"=>"1449033606215"}
  Game Load (516.0ms)  SELECT `games`.* FROM `games`
  Champion Load (1.0ms)  SELECT `champions`.* FROM `champions`
Completed 200 OK in 16455ms (Views: 1.0ms | ActiveRecord: 521.0ms)
```

Checked out testBranch (8 minutes ago)

Figure 8.2: Pre-server-side caching repeat call.

Subsequent calls to /champions only takes 6ms, a 99.9% improvement!



```
Started GET "/assets/logo-09aef6a2448229f7669d34614ee7f113cccf0f48c27a6cc34877c077c4d34b83.jpg" for 127.0.0.1 at 2015-12-01 21:37:09 -0800


Started GET "/assets/background-a4ba1606dec7339710ff107e61ab42285b36b3ef509011eda6f0080843229e9b.png" for 127.0.0.1 at 2015-12-01 21:37:09 -0800


Started GET "/champions/json?_=1449034629455" for 127.0.0.1 at 2015-12-01 21:37:09 -0800
Processing by ChampionsController#retrieveAllChampStats as JSON
  Parameters: {"_"=>"1449034629455"}
Completed 200 OK in 6ms (Views: 0.0ms | ActiveRecord: 0.0ms)
```

Checked out master (8 minutes ago)

93:49 CRLF windows-1252 Git: master

Figure 8.3: Post-server-side caching repeat call.

With the addition of server side caching, we saw a respectable improvement in our tsung tests as well.

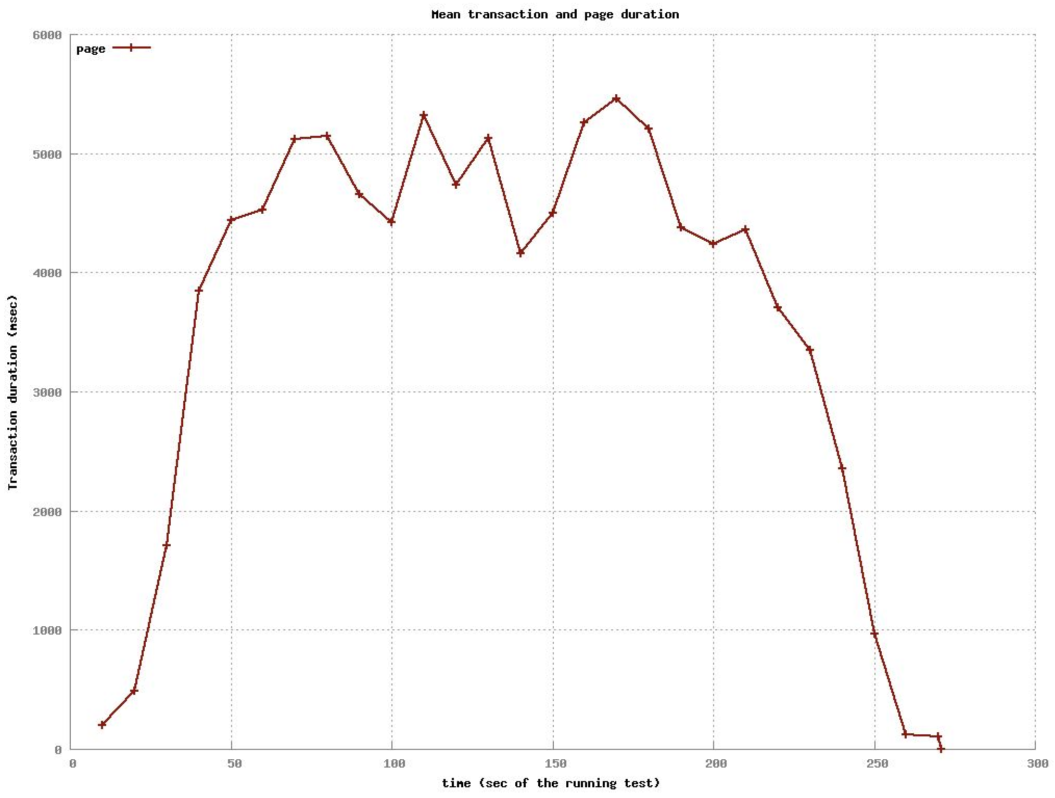
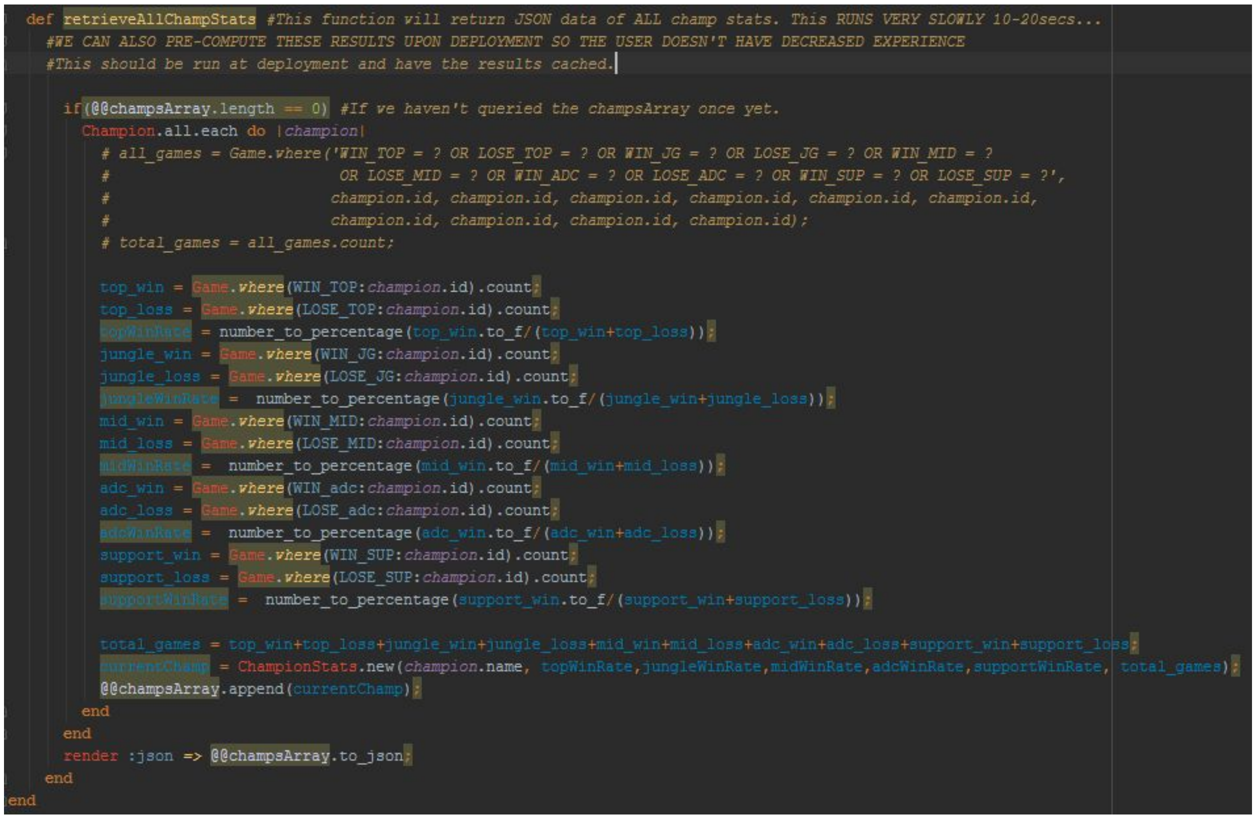


Figure 8.4: Mean transaction and page duration of tsung test on the application with server side caching

From the graph, we can see that the response rate decreased significantly from the unoptimized version. The mean response time dropped to 3.98 seconds and this saved the users an average of 1.2 minutes in total compared to the unoptimized app users. This improvement has made our app more usable, although the speed is still not yet acceptable in today's standards.

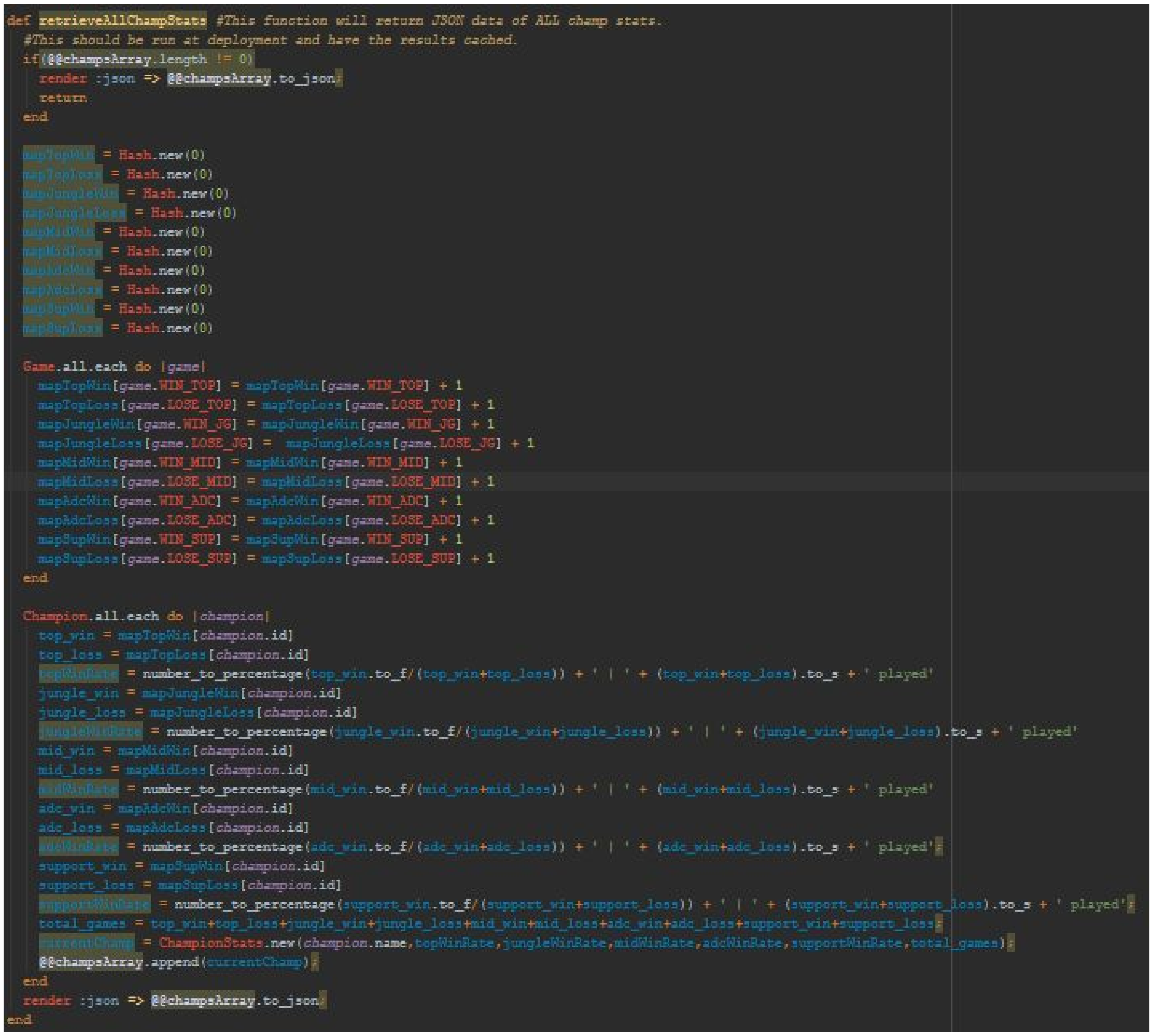
9 Optimization: MySQL Query Improvements



```
def retrieveAllChampStats #This function will return JSON data of ALL champ stats. This RUNS VERY SLOWLY 10-20secs...
  #WE CAN ALSO PRE-COMPUTE THESE RESULTS UPON DEPLOYMENT SO THE USER DOESN'T HAVE DECREASED EXPERIENCE
  #This should be run at deployment and have the results cached.

    if(@@champsArray.length == 0) #If we haven't queried the champsArray once yet.
      Champion.all.each do |champion|
        # all_games = Game.where('WIN_TOP = ? OR LOSE_TOP = ? OR WIN_JG = ? OR LOSE_JG = ? OR WIN_MID = ?
        #                          OR LOSE_MID = ? OR WIN_ADC = ? OR LOSE_ADC = ? OR WIN_SUP = ? OR LOSE_SUP = ?',
        #                         champion.id, champion.id, champion.id, champion.id, champion.id, champion.id,
        #                         champion.id, champion.id, champion.id, champion.id);
        # total_games = all_games.count;

        top_win = Game.where(WIN_TOP:champion.id).count;
        top_loss = Game.where(LOSE_TOP:champion.id).count;
        topWinRate = number_to_percentage(top_win.to_f/(top_win+top_loss));
        jungle_win = Game.where(WIN_JG:champion.id).count;
        jungle_loss = Game.where(LOSE_JG:champion.id).count;
        jungleWinRate =  number_to_percentage(jungle_win.to_f/(jungle_win+jungle_loss));
        mid_win = Game.where(WIN_MID:champion.id).count;
        mid_loss = Game.where(LOSE_MID:champion.id).count;
        midWinRate =  number_to_percentage(mid_win.to_f/(mid_win+mid_loss));
        adc_win = Game.where(WIN_adc:champion.id).count;
        adc_loss = Game.where(LOSE_adc:champion.id).count;
        adcWinRate =  number_to_percentage(adc_win.to_f/(adc_win+adc_loss));
        support_win = Game.where(WIN_SUP:champion.id).count;
        support_loss = Game.where(LOSE_SUP:champion.id).count;
        supportWinRate =  number_to_percentage(support_win.to_f/(support_win+support_loss));

        total_games = top_win+top_loss+jungle_win+jungle_loss+mid_win+mid_loss+adc_win+adc_loss+support_win+support_loss;
        currentChamp = ChampionStats.new(champion.name, topWinRate,jungleWinRate,midWinRate,adcWinRate,supportWinRate, total_games);
        @@champsArray.append(currentChamp);
      end
    end
    render :json => @@champsArray.to_json;
  end
end
```

Figure 9.1: Un-optimized MySQL query

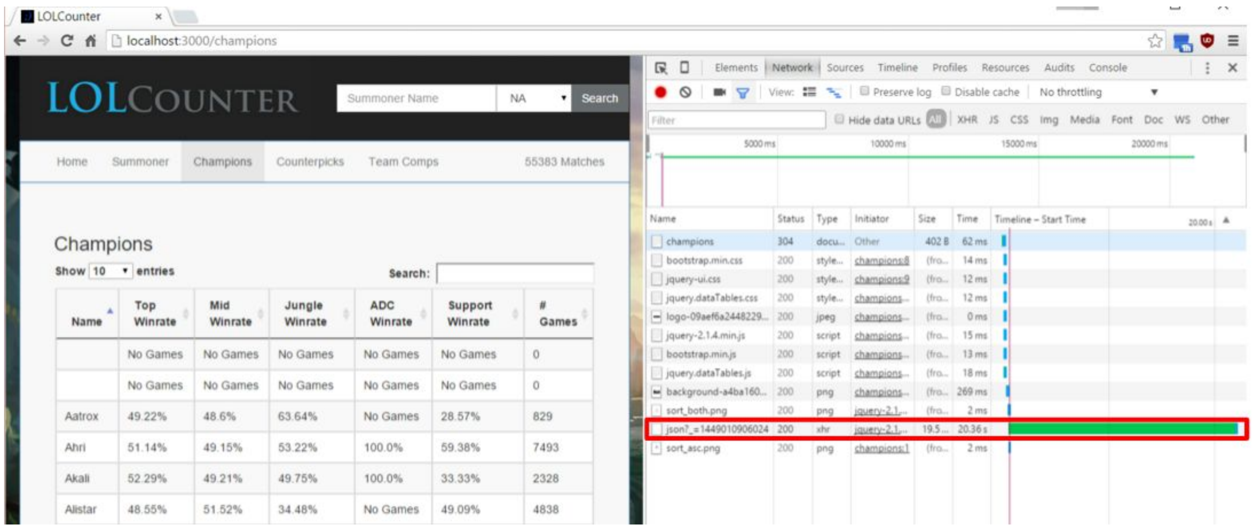
Originally, we used the above query to calculate win-rates of all champions from the matches database. This query does 10 DB scans per champion, resulting in 1280 DB scans which is terrible performance. After we identified this was the problem, we began working to write a more efficient query lookup.



```ruby
def retrieveAllChampStats #This function will return JSON data of ALL champ stats.
  #This should be run at deployment and have the results cached.
  if(@@champsArray.length != 0)
    render :json => @@champsArray.to_json;
    return
  end

  mapTopWin = Hash.new(0)
  mapTopLoss = Hash.new(0)
  mapJungleWin = Hash.new(0)
  mapJungleLoss = Hash.new(0)
  mapMidWin = Hash.new(0)
  mapMidLoss = Hash.new(0)
  mapAdcWin = Hash.new(0)
  mapAdcLoss = Hash.new(0)
  mapSupWin = Hash.new(0)
  mapSupLoss = Hash.new(0)

  Game.all.each do |game|
    mapTopWin[game.WIN_TOP] = mapTopWin[game.WIN_TOP] + 1
    mapTopLoss[game.LOSE_TOP] = mapTopLoss[game.LOSE_TOP] + 1
    mapJungleWin[game.WIN_JG] = mapJungleWin[game.WIN_JG] + 1
    mapJungleLoss[game.LOSE_JG] =  mapJungleLoss[game.LOSE_JG] + 1
    mapMidWin[game.WIN_MID] = mapMidWin[game.WIN_MID] + 1
    mapMidLoss[game.LOSE_MID] = mapMidLoss[game.LOSE_MID] + 1
    mapAdcWin[game.WIN_ADC] = mapAdcWin[game.WIN_ADC] + 1
    mapAdcLoss[game.LOSE_ADC] = mapAdcLoss[game.LOSE_ADC] + 1
    mapSupWin[game.WIN_SUP] = mapSupWin[game.WIN_SUP] + 1
    mapSupLoss[game.LOSE_SUP] = mapSupLoss[game.LOSE_SUP] + 1
  end

  Champion.all.each do |champion|
    top_win = mapTopWin[champion.id]
    top_loss = mapTopLoss[champion.id]
    topWinRate = number_to_percentage(top_win.to_f/(top_win+top_loss)) + ' | ' + (top_win+top_loss).to_s + ' played'
    jungle_win = mapJungleWin[champion.id]
    jungle_loss = mapJungleLoss[champion.id]
    jungleWinRate = number_to_percentage(jungle_win.to_f/(jungle_win+jungle_loss)) + ' | ' + (jungle_win+jungle_loss).to_s + ' played'
    mid_win = mapMidWin[champion.id]
    mid_loss = mapMidLoss[champion.id]
    midWinRate = number_to_percentage(mid_win.to_f/(mid_win+mid_loss)) + ' | ' + (mid_win+mid_loss).to_s + ' played'
    adc_win = mapAdcWin[champion.id]
    adc_loss = mapAdcLoss[champion.id]
    adcWinRate = number_to_percentage(adc_win.to_f/(adc_win+adc_loss)) + ' | ' + (adc_win+adc_loss).to_s + ' played';
    support_win = mapSupWin[champion.id]
    support_loss = mapSupLoss[champion.id]
    supportWinRate = number_to_percentage(support_win.to_f/(support_win+support_loss)) + ' | ' + (support_win+support_loss).to_s + ' played';
    total_games = top_win+top_loss+jungle_win+jungle_loss+mid_win+mid_loss+adc_win+adc_loss+support_win+support_loss;
    currentChamp = ChampionStats.new(champion.name,topWinRate,jungleWinRate,midWinRate,adcWinRate,supportWinRate,total_games);
    @@champsArray.append(currentChamp);
  end
  render :json => @@champsArray.to_json;
end
```

Figure 9.2: Updated MySQL query

By taking advantage of the fast insertion/lookup properties of Hash tables and cheap memory, we can now retrieve all the information we need using only 1 DB scan. The performance of the new implementation above is reflected in our data shown below.



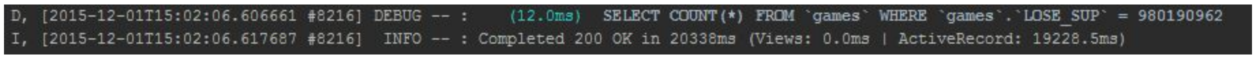
Champions

Show 10 entries Search:

Name	Top Winrate	Mid Winrate	Jungle Winrate	ADC Winrate	Support Winrate	# Games
	No Games	No Games	No Games	No Games	No Games	0
	No Games	No Games	No Games	No Games	No Games	0
Aatrox	49.22%	48.6%	63.64%	No Games	28.57%	829
Ahri	51.14%	49.15%	53.22%	100.0%	59.38%	7493
Akali	52.29%	49.21%	49.75%	100.0%	33.33%	2328
Alistar	48.55%	51.52%	34.48%	No Games	49.09%	4838

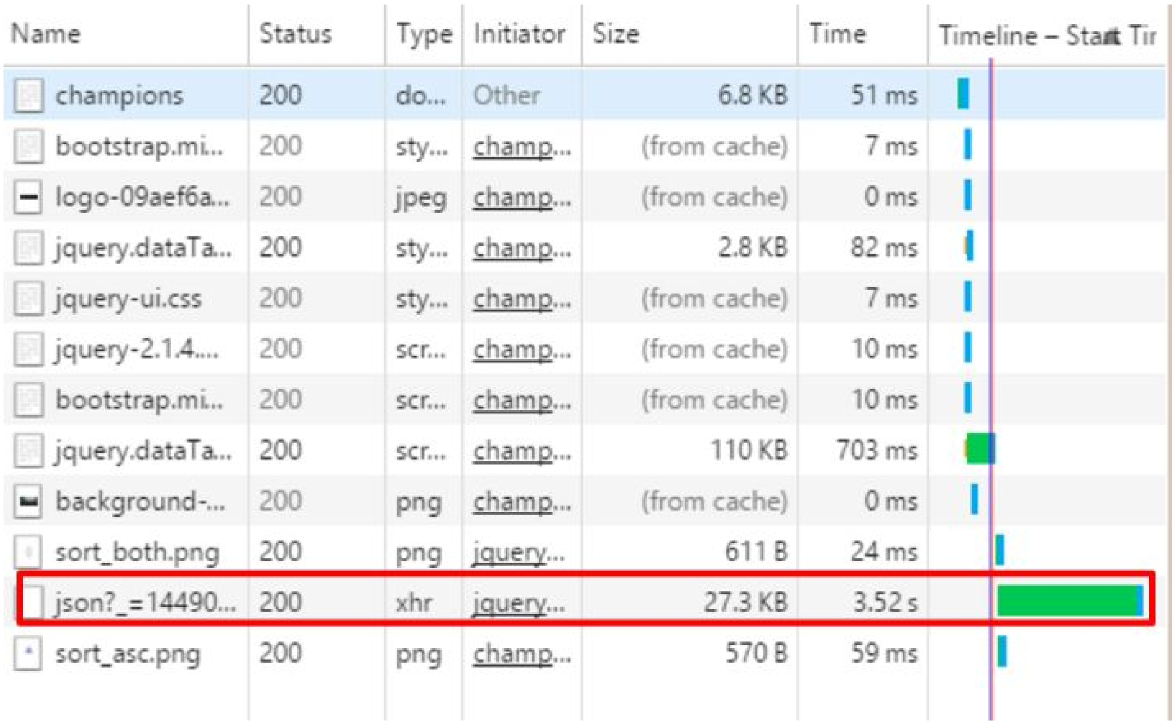
Name	Status	Type	Initiator	Size	Time	Timeline – Start Time
champions	304	docu...	Other	402 B	62 ms	
bootstrap.min.css	200	style...	champions:8	(fro...	14 ms	
jquery-ui.css	200	style...	champions:9	(fro...	12 ms	
jquery.dataTables.css	200	style...	champions...	(fro...	12 ms	
logo-09aef6a2448229...	200	jpeg	champions...	(fro...	0 ms	
jquery-2.1.4.min.js	200	script	champions...	(fro...	15 ms	
bootstrap.min.js	200	script	champions...	(fro...	13 ms	
jquery.dataTables.js	200	script	champions...	(fro...	18 ms	
background-a4ba160...	200	png	champions...	(fro...	269 ms	
sort_both.png	200	png	jquery-2.1...	(fro...	2 ms	
json?_=1449010906024	200	xhr	jquery-2.1...	19.5...	20.36 s	
sort_asc.png	200	png	champions:1	(fro...	2 ms	

Figure 9.3: Initial load time for champions table (this is before mysql optimizations)



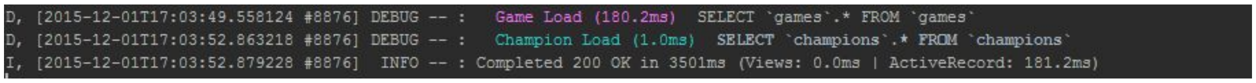
```
D, [2015-12-01T15:02:06.606661 #8216] DEBUG -- :    (12.0ms)  SELECT COUNT(*) FROM `games` WHERE `games`.`LOSE_SUP` = 980190962
I, [2015-12-01T15:02:06.617687 #8216]  INFO -- : Completed 200 OK in 20338ms (Views: 0.0ms | ActiveRecord: 19228.5ms)
```

Figure 9.4: Initial load time for champions table (this is before mysql optimizations)



Name	Status	Type	Initiator	Size	Time	Timeline – Start Tir
champions	200	do...	Other	6.8 KB	51 ms	
bootstrap.mi...	200	sty...	champ...	(from cache)	7 ms	
logo-09aef6a...	200	jpeg	champ...	(from cache)	0 ms	
jquery.dataTa...	200	sty...	champ...	2.8 KB	82 ms	
jquery-ui.css	200	sty...	champ...	(from cache)	7 ms	
jquery-2.1.4....	200	scr...	champ...	(from cache)	10 ms	
bootstrap.mi...	200	scr...	champ...	(from cache)	10 ms	
jquery.dataTa...	200	scr...	champ...	110 KB	703 ms	
background-...	200	png	champ...	(from cache)	0 ms	
sort_both.png	200	png	jquery...	611 B	24 ms	
json?_=14490...	200	xhr	jquery...	27.3 KB	3.52 s	
sort_asc.png	200	png	champ...	570 B	59 ms	

Figure 9.5: Initial load time for champions table after mysql optimizations



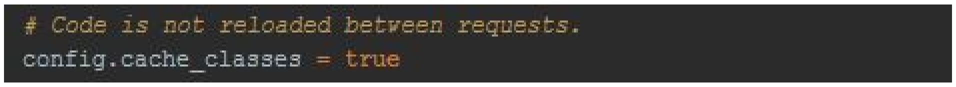
```
D, [2015-12-01T17:03:49.558124 #8876] DEBUG -- :   Game Load (180.2ms)  SELECT `games`.* FROM `games`
D, [2015-12-01T17:03:52.863218 #8876] DEBUG -- :   Champion Load (1.0ms)  SELECT `champions`.* FROM `champions`
I, [2015-12-01T17:03:52.879228 #8876]  INFO -- : Completed 200 OK in 3501ms (Views: 0.0ms | ActiveRecord: 181.2ms)
```

Figure 9.6: Initial load time for champions table after mysql optimizations

Although mysql optimization improved the speed of generating the champions table significantly, the true benefits are only noticeable for the very first user because server side caching removed the necessity to create the champions table more than once during the apps lifetime. Because of this, the tsung test resulted in speeds almost identical to an app without server side caching (mean of 3.96 seconds per request compared to the 3.98 seconds without the server side caching).

10 Optimization: client side caching

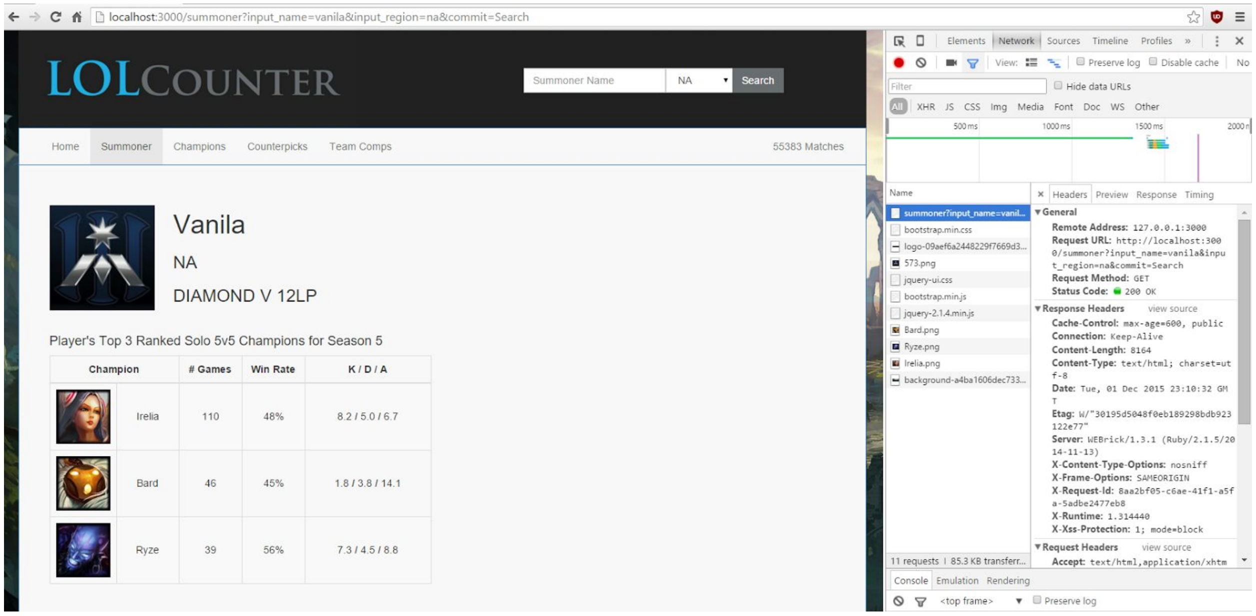
Next, we applied client-side optimizations to LoLCounter. When making a request to a page, the time needed can be divided into two intervals: Making the call to the server to request the page, and loading the page's contents for the user. Client-side caching speeds up the latter segment by storing caching data such as images and text in the user's browser so that repeated requests by the user can be accessed immediately. We first enabled Ruby on Rails' automatic static page caching in the production build, as seen below.



```
# Code is not reloaded between requests.
config.cache_classes = true
```

Figure 10.1: /config/environments.production.rb

Assume the user searches for a summoner name from the "Summoner" page. He types in his name, 'Vanila," and selects a region, North America, and then presses search. This request creates the following result:



Vanila

NA

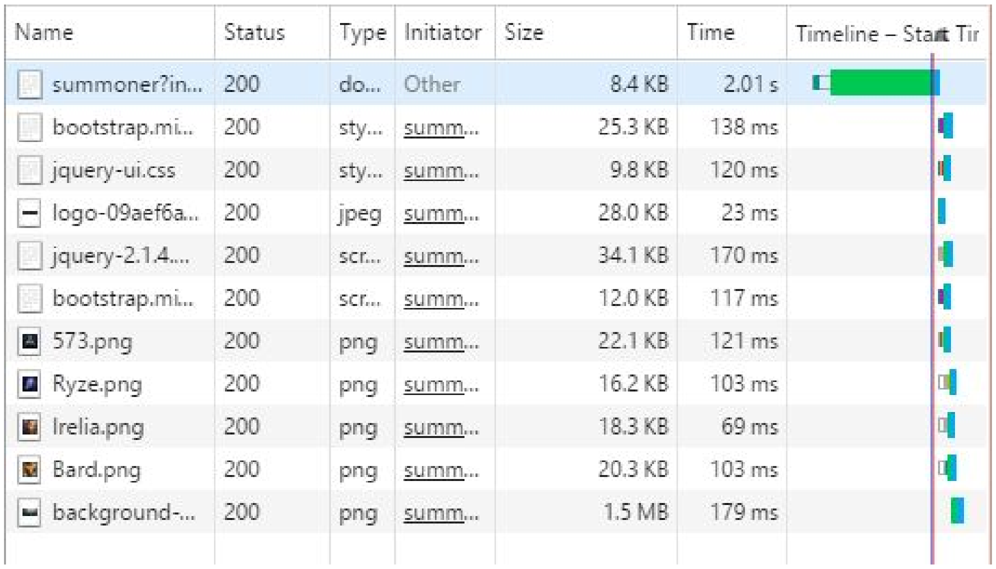
DIAMOND V 12LP

Player's Top 3 Ranked Solo 5v5 Champions for Season 5

Champion		# Games	Win Rate	K / D / A
	Irelia	110	48%	8.2 / 5.0 / 6.7
	Bard	46	45%	1.8 / 3.8 / 14.1
	Ryze	39	56%	7.3 / 4.5 / 8.8

```
▼General
   Remote Address: 127.0.0.1:3000
   Request URL: http://localhost:3000/summoner?input_name=vanila&input_region=na&commit=Search
   Request Method: GET
   Status Code: 200 OK
▼Response Headers      view source
   Cache-Control: max-age=600, public
   Connection: Keep-Alive
   Content-Length: 8164
   Content-Type: text/html; charset=utf-8
   Date: Tue, 01 Dec 2015 23:10:32 GMT
   Etag: W/"30195d5048f0eb189298bdb923122e77"
   Server: WEBrick/1.3.1 (Ruby/2.1.5/2014-11-13)
   X-Content-Type-Options: nosniff
   X-Frame-Options: SAMEORIGIN
   X-Request-Id: 8aa2bf05-c6ae-41f1-a5fa-5adbe2477eb8
   X-Runtime: 1.314440
   X-Xss-Protection: 1; mode=block
▼Request Headers      view source
   Accept: text/html,application/xhtm
```

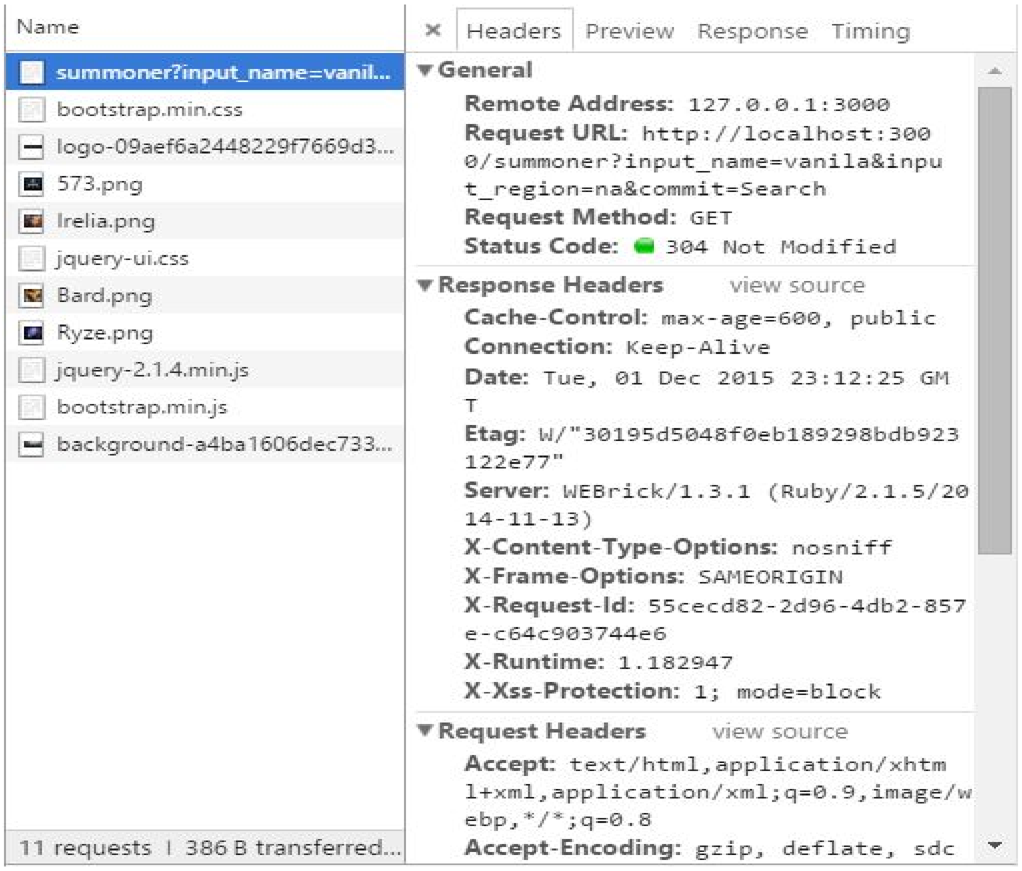
Figure 10.2: http://localhost:3000/summoner?input_name=vanila&input_region=na&commit=Search



Name	Status	Type	Initiator	Size	Time	Timeline – Start Tir
summoner?in...	200	do...	Other	8.4 KB	2.01 s	
bootstrap.mi...	200	sty...	summ...	25.3 KB	138 ms	
jquery-ui.css	200	sty...	summ...	9.8 KB	120 ms	
logo-09aef6a...	200	jpeg	summ...	28.0 KB	23 ms	
jquery-2.1.4....	200	scr...	summ...	34.1 KB	170 ms	
bootstrap.mi...	200	scr...	summ...	12.0 KB	117 ms	
573.png	200	png	summ...	22.1 KB	121 ms	
Ryze.png	200	png	summ...	16.2 KB	103 ms	
Irelia.png	200	png	summ...	18.3 KB	69 ms	
Bard.png	200	png	summ...	20.3 KB	103 ms	
background-...	200	png	summ...	1.5 MB	179 ms	

Figure 10.3: Initial page load time

The key part to consider is the response status code from figure 8.2: 200 OK GET. This means the outputted page was successfully fetched from the server and displayed. As seen in figure 8.3, it takes about 2 seconds to load the page. Next, our user searches with the same exact parameters again.

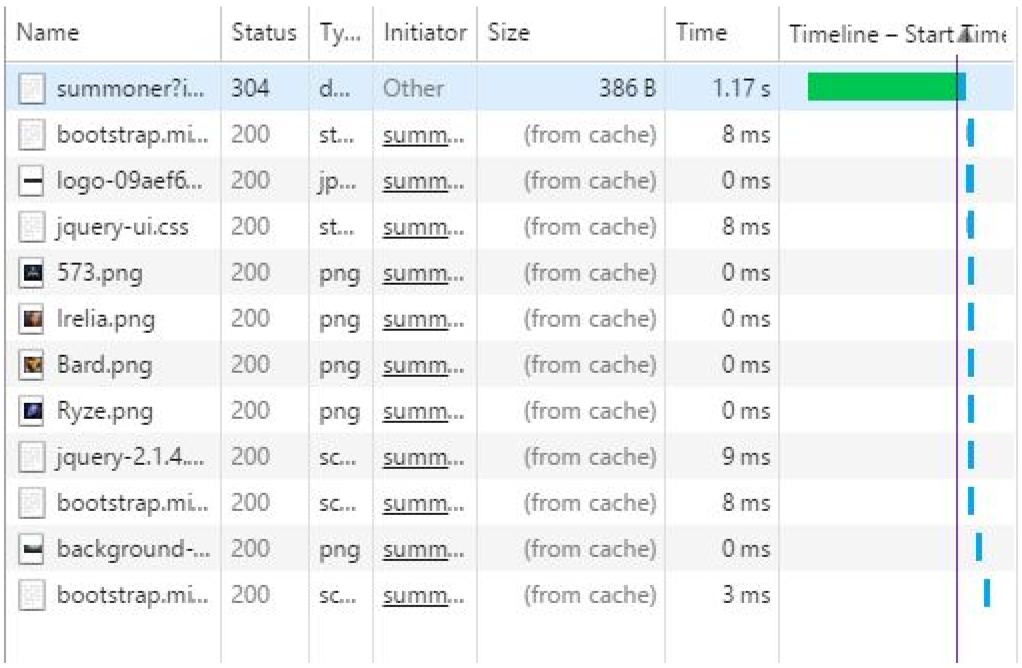


```
Name
summoner?input_name=vanil...
bootstrap.min.css
logo-09aef6a2448229f7669d3...
573.png
Irelia.png
jquery-ui.css
Bard.png
Ryze.png
jquery-2.1.4.min.js
bootstrap.min.js
background-a4ba1606dec733...
11 requests | 386 B transferred...
```

```
× Headers  Preview  Response  Timing
▼General
    Remote Address: 127.0.0.1:3000
    Request URL: http://localhost:300
    0/summoner?input_name=vanila&inpu
    t_region=na&commit=Search
    Request Method: GET
    Status Code: ● 304 Not Modified
▼Response Headers      view source
    Cache-Control: max-age=600, public
    Connection: Keep-Alive
    Date: Tue, 01 Dec 2015 23:12:25 GM
    T
    Etag: W/"30195d5048f0eb189298bdb923
    122e77"
    Server: WEBrick/1.3.1 (Ruby/2.1.5/20
    14-11-13)
    X-Content-Type-Options: nosniff
    X-Frame-Options: SAMEORIGIN
    X-Request-Id: 55cecd82-2d96-4db2-857
    e-c64c903744e6
    X-Runtime: 1.182947
    X-Xss-Protection: 1; mode=block
▼Request Headers      view source
    Accept: text/html,application/xhtm
    l+xml,application/xml;q=0.9,image/w
    ebp,*/*;q=0.8
    Accept-Encoding: gzip, deflate, sdc
```

Figure 10.4: Repeated request as in figure 5.1 results in status code 304 Not Modified

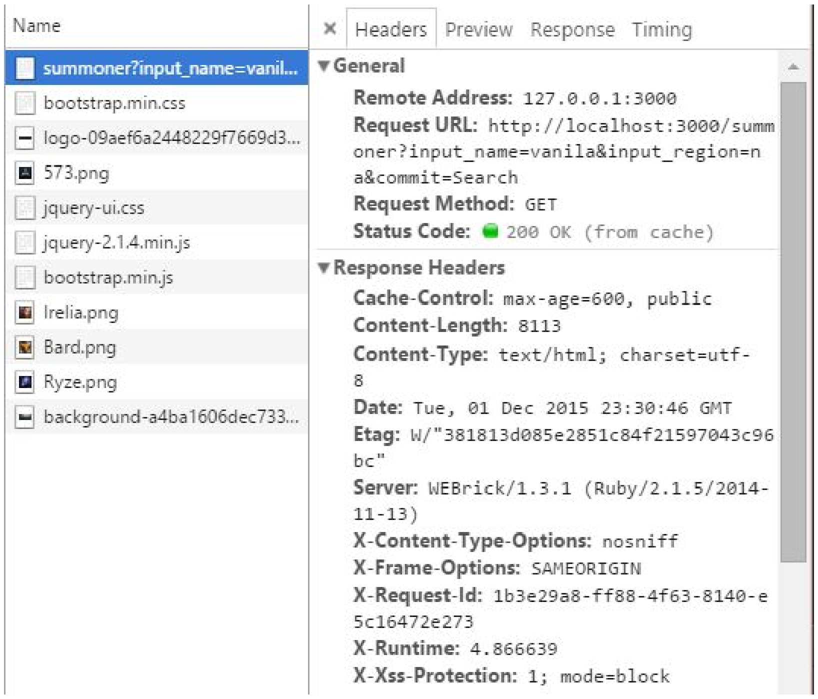
As seen by the Status Code, 304 Not Modified was returned instead of 200. A response code of 304 means Rails recognized that the second search was requesting data that had been previously searched, and hence displays that data instead of making a new request to the server. Furthermore, static files get served from cache, as seen below:



Name	Status	Ty...	Initiator	Size	Time	Timeline – Start Time
summoner?i...	304	d...	Other	386 B	1.17 s	
bootstrap.mi...	200	st...	summ...	(from cache)	8 ms	
logo-09aef6...	200	jp...	summ...	(from cache)	0 ms	
jquery-ui.css	200	st...	summ...	(from cache)	8 ms	
573.png	200	png	summ...	(from cache)	0 ms	
Irelia.png	200	png	summ...	(from cache)	0 ms	
Bard.png	200	png	summ...	(from cache)	0 ms	
Ryze.png	200	png	summ...	(from cache)	0 ms	
jquery-2.1.4....	200	sc...	summ...	(from cache)	9 ms	
bootstrap.mi...	200	sc...	summ...	(from cache)	8 ms	
background-...	200	png	summ...	(from cache)	0 ms	
bootstrap.mi...	200	sc...	summ...	(from cache)	3 ms	

Figure 10.5: Repeated search request loads static files from cache, but the request is still executed.

While this is a slight optimization, we can do better than simply turning caching on. Rails also provides time-based cache headers. One of these headers is “expires_in,” which sets a max-age parameter for how long a page is cached in the browser. For the “Summoners” page, we set a max-age value of 10 minutes, which is the minimum likely time for a user’s match history on Riot to update. Now, when we repeat a search request, we receive a new response.

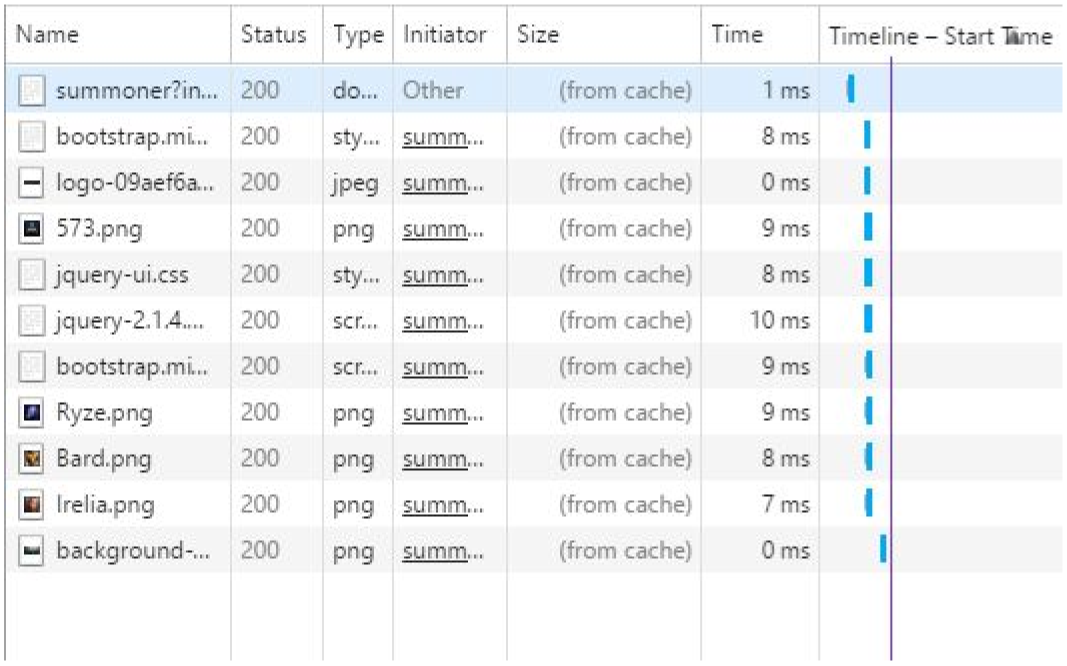


Name
summoner?input_name=vanil...
bootstrap.min.css
logo-09aef6a2448229f7669d3...
573.png
jquery-ui.css
jquery-2.1.4.min.js
bootstrap.min.js
Irelia.png
Bard.png
Ryze.png
background-a4ba1606dec733...

```
▼General
    Remote Address: 127.0.0.1:3000
    Request URL: http://localhost:3000/summ
    oner?input_name=vanila&input_region=n
    a&commit=Search
    Request Method: GET
    Status Code: ● 200 OK (from cache)
▼Response Headers
    Cache-Control: max-age=600, public
    Content-Length: 8113
    Content-Type: text/html; charset=utf-
    8
    Date: Tue, 01 Dec 2015 23:30:46 GMT
    Etag: W/"381813d085e2851c84f21597043c96
    bc"
    Server: WEBrick/1.3.1 (Ruby/2.1.5/2014-
    11-13)
    X-Content-Type-Options: nosniff
    X-Frame-Options: SAMEORIGIN
    X-Request-Id: 1b3e29a8-ff88-4f63-8140-e
    5c16472e273
    X-Runtime: 4.866639
    X-Xss-Protection: 1; mode=block
```

Figure 10.6: Repeated search request after "expires_in" header implemented results in 200 OK (from cache)

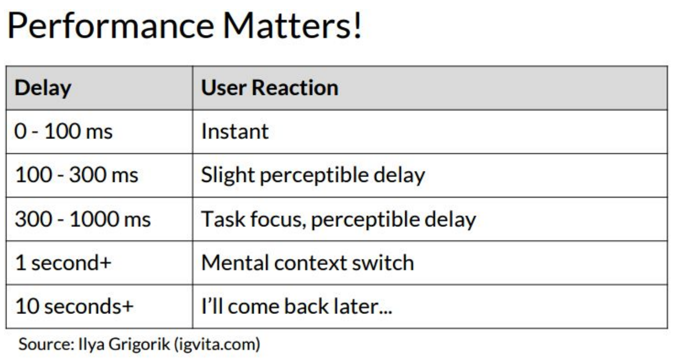
This new header means the request also gets cached. What does this say for our page load time?



Name	Status	Type	Initiator	Size	Time	Timeline – Start Time
summoner?in...	200	do...	Other	(from cache)	1 ms	
bootstrap.mi...	200	sty...	summ...	(from cache)	8 ms	
logo-09aef6a...	200	jpeg	summ...	(from cache)	0 ms	
573.png	200	png	summ...	(from cache)	9 ms	
jquery-ui.css	200	sty...	summ...	(from cache)	8 ms	
jquery-2.1.4....	200	scr...	summ...	(from cache)	10 ms	
bootstrap.mi...	200	scr...	summ...	(from cache)	9 ms	
Ryze.png	200	png	summ...	(from cache)	9 ms	
Bard.png	200	png	summ...	(from cache)	8 ms	
Irelia.png	200	png	summ...	(from cache)	7 ms	
background-...	200	png	summ...	(from cache)	0 ms	

Figure 10.7: Repeated request using "expires_in" header results in a nearly instantaneous page load.

The page loads almost immediately! This is because the search request was immediately recognized as a repeat and there was no need to communicate with the server. This means that every repeated search request will save 2 seconds of search and load time within 10 minutes of the initial search. While this may seem miniscule, every second we shave off the page load time improves user experience, especially since user perception is based on such small proportions.



Performance Matters!

Delay	User Reaction
0 - 100 ms	Instant
100 - 300 ms	Slight perceptible delay
300 - 1000 ms	Task focus, perceptible delay
1 second+	Mental context switch
10 seconds+	I'll come back later...

Source: Ilya Grigorik (igvita.com)

Figure 10.8: Prof. Mutz's slides on Client-Side Caching

Adding the client side caching also caused the most dramatic increase in scalability, as shown by the tsung test.

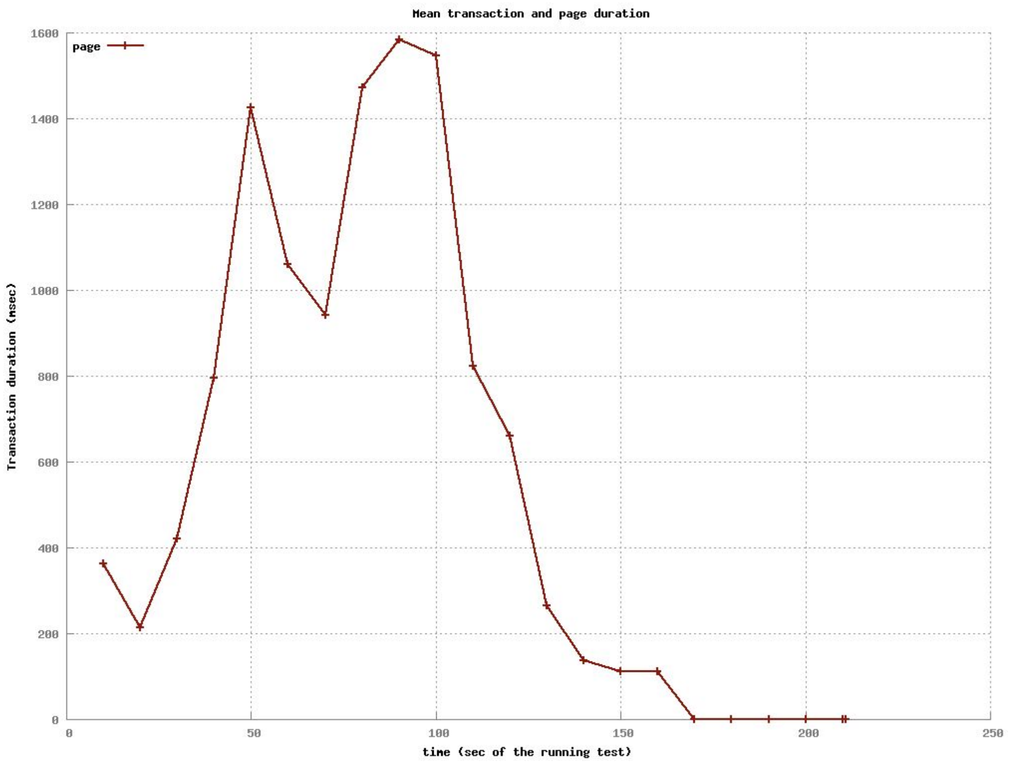


Figure 10.9: Mean transaction and page duration of tsung test on fully optimized application

As we can see, most transactions look less than 2 seconds, which is almost double the speed from before. Each user session took half as long (about 2 minutes) and all transactions were at an acceptable range. By adding static caching, we made all repeated visits almost instantaneous and reduced the load on our database significantly. Without the client side caching, at around the 50 second mark, the transaction duration would have continued to increase. Instead, the transaction time actually decreased because clients were starting to revisit pages that are cached. The transaction duration started increasing again because the total number of users grew large enough that its load was more significant than the benefits of caching.

11 Vertical Scaling

Our final optimization implemented vertical scaling by using a better AWS server. Instead of deploying the app on an m1.medium instance, we used a m1.xlarge instance.

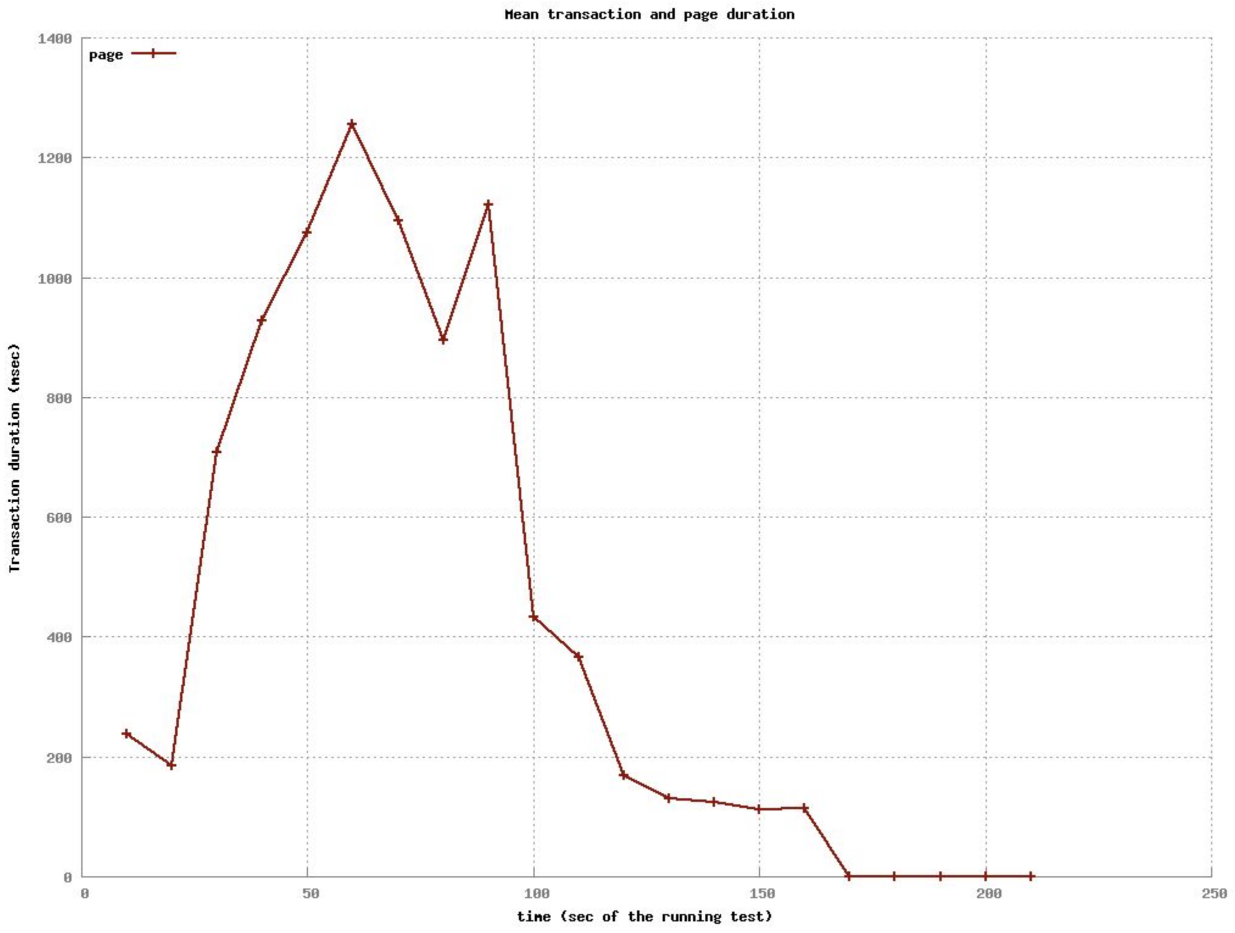


figure 11.1: Mean transaction and page duration of tsung test on optimized application running on m1.xlarge instance

As expected, the resulting tsung test ran even faster. The average transaction took about 0.2 seconds less than the app running on m1.medium instance. This was a simple demonstration of how vertical scaling can also improve our application, even though it was not the focus of our optimizations.

12 Optimization wishlist

The most significant bottleneck of our application is the database. Because our database could not be modified by the users, we hoped to create multiple identical databases and set up a load balancer between the app server and the databases. Unfortunately, this was deemed not viable for our project due to the difficult constraints of Amazon Web Service.

We also hoped to implement horizontal scaling using multiple servers and based on location. This would have been a reasonable next step because League of Legends is naturally divided by region. If we assumed that our users were on a national or a global level, it would have been a necessary step but it went beyond the scope of our project needs.

13 Things we learned

In terms of technical knowledge, we walked away from this project with many new assets. We now know how to code in Ruby and how to create web applications using Rails. We learned how to use tools such as the developer console and Tsung to test our optimizations. We gained experience using relational databases to store massive amounts of data.

On the other hand, we picked up a variety of techniques related to software development. Our first two meetings involved us discussing the features we wanted to implement: There are a lot of features we could have implemented, but would they really have been useful to our (imaginary) users? We didn’t want to make something that was cool but absolutely useless. That was why we settled on champion win rates, counterpick win rates, and team composition win rates. We continued following this methodology in designing our app. Despite knowing that nobody would ever use our application (Our AWS instance would end when the course finishes), we designed with the user in mind: For example, in the front-end, we made sure our elements were responsive and scaled with page size. Our app is even mobile and tablet-friendly. We also added multiple catches for user input, such as returning errors for incorrectly spelled summoner or champion names, and a very informative homepage that includes embedded video and game-related content.

14 Further development

We succeeded in implementing all of LoLCounter's features that we set out to do. However, to create a data aggregation site that can rival existing third-party sites such as *LolKing.net*, *OP.gg*, and *Champion.gg*, we need our match data to enter the tens of millions. For example, *LeagueofGraphs.com* currently displays the results of 20,880,000 games. Currently, we generate our games database on the side and manually load it into LoLCounter. In order for us to hit millions of games, this process must be automated. The reason we chose not to implement this feature this quarter was because we knew that when our AWS instance would terminate, we would lose all our gathered matches. Hence, we chose to manually load in a 300k match database every time.

15 Conclusion

Over the course of ten weeks, we created a minimum viable product that successfully analyzes hundreds of thousands of League of Legends games. This experience taught us the value of group software development and how to distribute production of large applications. We also learned of the scaling problems facing large web applications and how they can be solved, and applied some of those techniques to our own project. In LoLCounter, we successfully implemented server-side caching, client-side caching, MySQL optimizations, vertical scaling, and analyzed how each improved page responsiveness. Compared to our baseline which has no scaling vs our final product, we can see each user notices an increase in page responsiveness by about 4 - 6 seconds, or a 57% - 86% improvement. We believe if we apply horizontal scaling and database sharding, the improvement will be even more. Scalability is a big topic in modern software development, and this course definitely prepared us in our future as software developers.