

---

# Scaling a Web-Based Tutoring System

From Classrooms to MOOCs

---

*A thesis submitted in fulfillment of the requirements  
for the degree of Master of Technology*

*by*

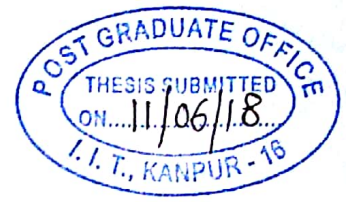
**Akshay Jindal**

**(16111029)**



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July 2018



## Certificate

It is certified that the work contained in this thesis entitled "*Scaling a Web-Based Tutoring System: From Classrooms to MOOCs*" by *Akshay Jindal* has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

*Arnab Bhattacharya*

---

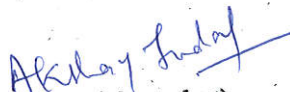
Dr. Arnab Bhattacharya

*June 2018*

Associate Professor,  
Department of Computer Science and Engineering  
Indian Institute of Technology Kanpur

### Statement of Thesis Preparation

1. Thesis title: Scaling a Web-Based Tutoring System: From Classroom to MOOCs
2. Degree for which the thesis is submitted: M.Tech
3. Thesis Guide was referred to for preparing the thesis.
4. Specifications regarding thesis format have been closely followed.
5. The contents of the thesis have been organized based on the guidelines.
6. The thesis has been prepared without resorting to plagiarism.
7. All sources used have been cited appropriately.
8. The thesis has not been submitted elsewhere for a degree.

  
(Signature of the student)

Name: AKSHAY JINDAL  
Roll No.: 16111029  
Department/IDP: CSE

# *Abstract*

Massive Open Online Courses (MOOCs) have had an immense role to play in the arena of digital learning. Out of the various courses offered through the popular MOOC platforms, the introductory programming courses have been found to be heavily favored. In such courses, the enrollment reaches 5000-10000 students. Prutor is a tool developed at the Indian Institute of Technology Kanpur for introductory programming courses. It enables the instructors to conduct proctored lab exams and sessions at a classroom level, with real-time feedback. Currently, it runs at a course level, supporting around 400 students at a time. To work seamlessly in a MOOC environment, it should be able to handle an incoming load of at least 5000 active users, which is what we have tried to achieve in this thesis.

To achieve the scalability, we have analyzed each component in the technology stack of Prutor, which from a developer's perspective, is a cloud-based web application running on Docker containers, with the help of standard benchmarking and profiling tools. The analysis of logs generated by these components led us to tune some of the critical configuration parameters from their default values to more appropriate ones. We have also analyzed the database component of Prutor. Using standard tools, we have examined the most frequently run queries for join computations, index usage, full table scans, on-the-fly sorting, and creation of temporary tables. Following this analysis, we have experimented with altered indexes and various optimization techniques such as index hints in the query and materialized views. Prutor has two server components, namely Engine and WebApp. We have removed the additional database abstraction layer without compromising with the security in the engine component. This, in turn, has led to a drastic reduction in the number of queries executed on the database server per operation. We have also reduced the number of database hits in several APIs such as login, compile, execute and evaluate, which consequently gave better average response times. Using all these, we have successfully reached a scale of 3000 users on WebApp and 1000 users on Engine and 1000 users combined on an 8-core machine with Core i7 3.40 GHz processor.

## *Acknowledgements*

I would like to extend my sincerest gratitude to my thesis supervisor Dr. Arnab Bhattacharya for his unparalleled guidance and support. I am deeply grateful to Dr. Amey Karkare for always being open to discussions. I am genuinely thankful to them for allowing me to work at my own pace. I was new to this area, but still, they gave me full ownership of such an important project, which is and will keep on being used by generations of students of IIT Kanpur and other premier institutes in India. Their trust and patience inspired me to pull things off. This project has given me a rock-solid foundation in the area of Scalable Web Applications, which will surely help me in my career.

I would like to specially thank Mr. Adarsh Jagannatha for midnight CCD discussions, Mr. Ravi Shankar Mula for numerous google hangouts and Mr. Saurabh Verma for his code-walkthrough sessions on Prutor. Without their guidance, I would not have been able to even start this project. I would also like to thank my Friends Abhishek and Manish for reviewing my report. Thank you all for everything.

Last but not the least, without the unconditional support of my Family, I would not have been able to enter this institute, so a big cheers to my folks. I would also like to thank my Friends for standing by my side throughout my 2 years journey.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	2
1.2 Motivation . . . . .	2
1.3 Contributions of the thesis . . . . .	2
1.4 Organization of the thesis . . . . .	3
<b>2 Background, Tools and Related Work</b>	<b>5</b>
2.1 Prutor . . . . .	5
2.1.1 HTTP Reverse Proxy . . . . .	6
2.1.2 Relational Database . . . . .	7
2.1.3 NoSQL Database . . . . .	7
2.1.4 In-Memory cache . . . . .	7
2.1.5 WebApp . . . . .	7
2.1.6 Engine . . . . .	7
2.2 Tools used . . . . .	8
2.2.1 Apache JMeter . . . . .	8
2.2.2 New Relic APM . . . . .	9
2.2.3 MySQL Workbench . . . . .	10
2.2.4 ps_mem.py . . . . .	10
2.3 Related Work . . . . .	11
<b>3 WebApp: Analysis and Benchmarking</b>	<b>12</b>
3.1 Perspectives . . . . .	12

3.1.1	Student Activities in an Event	13
3.2	Baselining Scalability of WebApp	15
3.2.1	Distributed Setup	15
3.2.2	Workload Specification	17
3.2.3	Performance Metrics	17
3.2.4	Results	18
3.3	Performance Profiling	19
3.3.1	Setup	19
3.3.2	Profiling of each feature	19
3.3.3	Analysis of the common bottlenecks	23
3.3.3.1	Memcached	23
3.3.3.2	readFile	28
3.3.4	Error Analytics	28
3.3.4.1	HAProxy logs analysis	31
3.4	Incremental Modifications	32
3.4.1	Reaching 500 on 2	32
3.4.2	Reaching 1000 on 4	36
3.4.3	Reaching 2000 on 8	38
3.4.4	Reaching 3000 on 16	41
3.4.5	Comparative Results	42
<b>4</b>	<b>Database Analysis</b>	<b>43</b>
4.1	Query Analysis	43
4.1.1	Index Usage	44
4.1.2	Queries doing Sorting	48
4.1.3	Queries doing full table scans	49
4.1.4	Queries Using Temporary Tables	49
4.2	Usage of Materialized Views	50
4.2.1	ongoingAssignments_mv	50
4.2.2	codebook_mv	51
4.3	Comparative Results	52
4.4	Parameter Tuning	52
4.4.1	innodb_buffer_pool_size	53
4.4.2	innodb_buffer_pool_instances	53
4.4.3	Escape from Swap	54
4.4.4	Query Cache	54
4.4.5	skip_name_resolve	54
<b>5</b>	<b>Engine: Analysis and Benchmarking</b>	<b>55</b>
5.1	Event Workload	55
5.2	Baselining Scalability of Engine	56
5.2.1	Setup	56
5.2.2	Workload Specification	56
5.2.3	Performance Metrics	56

5.2.4	Results	57
5.3	Parameter Tuning	57
5.3.1	Reaching 500 users	58
5.3.2	Reaching 1000 users	60
5.4	Other modifications	61
5.4.1	ORM Removal	61
5.4.1.1	Extra Queries	62
5.4.1.2	Usage of PDO	63
5.4.2	Removal of unnecessary DB hits	64
5.4.3	Removal of NoSQL hits	65
5.5	Results	66
5.5.1	Reduction in Queries	66
5.5.2	Comparative Results	67
<b>6</b>	<b>End Result Performance Comparison</b>	<b>69</b>
6.1	Comparative Results of WebApp on 48 core machine	69
6.2	Comparative Results of Engine on 48 core machine	70
6.3	8 core i7 V/s 48 core xeon Comparison of WebApp	71
6.4	8 core i7 V/s 48 core xeon Comparison of Engine	73
6.5	Variation of WebApp Performance based on cores	75
6.6	Parameter Summary	75
6.6.1	Engine on 8 core	76
6.6.2	WebApp on 8 core	76
6.6.3	Engine on 48 core	77
6.6.4	WebApp on 48 core	77
6.6.5	Combined on 8 core	78
6.6.6	Combined on 48 core	78
<b>7</b>	<b>Conclusion and Future Work</b>	<b>79</b>
7.1	Conclusion	79
7.2	Future Work	80
	<b>Bibliography</b>	<b>84</b>
	<b>Appendices</b>	<b>85</b>
<b>A</b>	<b>Apache JMeter: Setup Instructions</b>	<b>86</b>
A.1	Apache JMeter: Setup Guide	86
A.1.1	Java Installation	86
A.1.2	JMeter Installation	86
A.1.2.1	Jargons of JMeter	88
A.1.3	Step-By-Step Construction of a test plan	90
A.2	Distributed Load Testing using JMeter	96

A.2.1	Distributed Test Setup:Step-by-Step . . . . .	96
<b>B</b>	<b>MySQL-WorkBench: Setup Instructions</b>	<b>99</b>
B.1	Build an Image of MySQL 5.7 . . . . .	99
B.2	Setting up MySQL-Workbench . . . . .	100
B.2.1	Enabling SSH on rdbupdated container . . . . .	100
B.2.2	Installing mysql-workbench . . . . .	101
<b>C</b>	<b>New-Relic: Setup Instructions</b>	<b>102</b>
C.1	New-Relic Servers for Linux . . . . .	102
C.2	Enable Monitoring for Docker . . . . .	103
C.3	Installing New-Relic Node.js agent . . . . .	103
<b>D</b>	<b>Instructions for Parameter Tuning</b>	<b>104</b>
D.1	Tuning Engine Parameters . . . . .	104
D.2	Tuning WebApp Parameters . . . . .	105
D.3	Tuning HAProxy . . . . .	105
<b>E</b>	<b>Repository links</b>	<b>106</b>
E.1	Prutor . . . . .	106
E.2	WebApp . . . . .	106
E.2.1	Steps to update Nodejs . . . . .	106
E.3	Engine . . . . .	106
E.3.1	Related to Sec. 5.4.3 . . . . .	107
E.4	Steps to Setup . . . . .	107
E.5	Link of all the important docs related to Thesis . . . . .	107
E.6	All the JMeter Test Plans . . . . .	108

# List of Figures

2.1	Prutor: Physical view . . . . .	6
2.2	Role of L7 Reverse Proxy(HAProxy) . . . . .	6
3.1	JMeter:Distributed Master-Slave Setup . . . . .	16
3.2	VUs v/s Error% for 2 webapps . . . . .	18
3.3	VUs v/s Error% for 4 webapps . . . . .	18
3.4	VUs v/s Error% for 8 webapps . . . . .	18
3.5	VUs v/s Error% for 16 webapps . . . . .	18
3.6	Comparison of Baseline Error% on different number of webapps . . . . .	19
3.7	View Assignments:Slowest Components . . . . .	20
3.8	View Assignments:Breakdown Table . . . . .	20
3.9	View Assignments:Custom Tracer . . . . .	21
3.10	View Home Page:Breakdown Table . . . . .	21
3.11	View Home Page:Slowest Components . . . . .	22
3.12	User Login:Breakdown Table . . . . .	22
3.13	Save Code:Slowest Components . . . . .	23
3.14	Save Code:Breakdown Table . . . . .	23
3.15	Memcached:Storage & Retrieval of Sessions . . . . .	24
3.16	Memcache: Slowest Components . . . . .	25
3.17	Memcache:Detailed Trace . . . . .	26
3.18	Memcache: netstat . . . . .	27
3.19	Slow component: readFile . . . . .	28
3.20	Error Analytics:Classification . . . . .	29
3.21	Error Analytics: part 1 . . . . .	30
3.22	Error Analytics: part 2 . . . . .	30
3.23	HAproxy:503 . . . . .	31
3.24	HAproxy:sQ . . . . .	31
3.25	HAproxy:sD . . . . .	31
3.26	Average Response Times before Login Optimization . . . . .	35
3.27	Average Response Times after Login Optimization . . . . .	35
3.28	Time taken by Bcrypt . . . . .	36
3.29	MySQL:500 connections exhausted . . . . .	40
3.30	Comparison of before and after for 2 webapps . . . . .	42
3.31	Comparison of before and after for 4 webapps . . . . .	42
3.32	Comparison of before and after for 8 webapps . . . . .	42

3.33	Comparison of before and after for 16 webapps . . . . .	42
4.1	Explain:getOngoingEvents . . . . .	44
4.2	Explain:Get Questions in the Ongoing Event . . . . .	45
4.3	Explain:getCourseStatistics . . . . .	45
4.4	Explain:getScoreCard . . . . .	46
4.5	Explain:getCodebook . . . . .	46
4.6	Explain:getLastSavedCode . . . . .	47
4.7	Explain:getAssignmentProblem . . . . .	47
4.8	Explain:getAssignmentDetails . . . . .	48
4.9	Explain:isAllowed . . . . .	48
4.10	Queries doing Sorting . . . . .	48
4.11	Queries doing full table scan . . . . .	49
4.12	Queries using temporary tables . . . . .	49
4.13	MV:ongoingAssignments . . . . .	50
4.14	MV:Trigger for updating ongoingAssignment MV . . . . .	51
4.15	MV:codebook . . . . .	51
5.1	Queries executed during single compilation request . . . . .	62
5.2	Queries executed during single execution request . . . . .	63
5.3	Queries executed during single evaluation request . . . . .	63
5.4	Extra MySQL Query: Fetch assignment . . . . .	65
5.5	Extra MySQL Query: Fetch visible test cases . . . . .	65
5.6	Extra MySQL Query: Fetch invisible testcases . . . . .	65
5.7	Queries executed during compilation after our changes . . . . .	66
5.8	Queries executed during execution after our changes . . . . .	66
5.9	Queries executed during evaluation after our changes . . . . .	66
5.10	Comparison of before and after for 10 Apache Workers . . . . .	67
5.11	Comparison of before and after for 20 Apache Workers . . . . .	67
5.12	Comparison of before and after for 4 engines . . . . .	67
5.13	Comparison of before and after for 8 engines . . . . .	67
5.14	Comparison of before and after for 16 engines . . . . .	67
5.15	Comparison of engine before and after our code changes . . . . .	68
6.1	Comparison of before and after for 1 Node Worker on 48 core machine . . . . .	69
6.2	Comparison of before and after for 2 Node Workers on 48 core machine . . . . .	69
6.3	Comparison of before and after for 4 Node Workers on 48 core machine . . . . .	70
6.4	Comparison of before and after for 8 Node Workers on 48 core machine . . . . .	70
6.5	Comparison of before and after for 16 Node Workers on 48 core machine . . . . .	70
6.6	Comparison of before and after for 24 Node Workers on 48 core machine . . . . .	70
6.7	Comparison of before and after for 10 Apache Worker on 48 core machine . . . . .	70
6.8	Comparison of before and after for 20 Apache Workers on 48 core machine . . . . .	70
6.9	Comparison of before and after for 40 Apache Workers on 48 core machine . . . . .	71
6.10	Comparison of before and after for 80 Apache Workers on 48 core machine . . . . .	71
6.11	Comparison of before and after for 160 Apache workers on 48 core machine . . . . .	71

---

6.12	Comparison of Node workers on 8 core i7 and 48 core xeon . . . . .	72
6.13	Comparison of WebApp Response Times on 8 core i7 and 48 core xeon . . . . .	73
6.14	Comparison of Apache Workers on 8 core i7 and 48 core xeon . . . . .	74
6.15	Comparison of Engine Response Times on 8 core i7 and 48 core xeon . . . . .	74
6.16	Comparison of Engine Response Times on 8 core i7 and 48 core xeon . . . . .	75
A.1	JMeter:Home Screen . . . . .	88
A.2	JMeter:Thread Group added . . . . .	89
A.3	JMeter:Final Test Plan . . . . .	90
A.4	JMeter:Thread Group properties . . . . .	92
A.5	JMeter:Login Request Added . . . . .	93
A.6	JMeter:Home Request Element . . . . .	94
A.7	JMeter:Save Code Request . . . . .	95

# List of Tables

3.1	User Login . . . . .	13
3.2	View Home . . . . .	13
3.3	View Codebook . . . . .	14
3.4	Open assignment . . . . .	14
3.5	Manually save code . . . . .	14
3.6	Submit Code . . . . .	14
3.7	100 requests on 1 Node worker with no tuning . . . . .	32
3.8	100 requests on 1 Node Worker with server maxconn tuned . . . . .	33
3.9	100 users 1 Node Worker with Cache removed . . . . .	34
3.10	200 users 2 workers on WebApp . . . . .	35
3.11	750 users 2 webapps server maxconn 500 . . . . .	37
3.12	750 users 2 workers server maxconn 1000 . . . . .	37
3.13	1000 users 4 Node workers . . . . .	38
3.14	1500 users on 4 workers . . . . .	38
3.15	1500 users on 4 workers server maxconn 1500 . . . . .	39
3.16	2000 users on 8 workers . . . . .	39
3.17	2000 users on 8 workers server maxconn 2000 . . . . .	40
3.18	2000 users on 8 Node workers . . . . .	41
3.19	3000 users on 16 Node workers . . . . .	41
4.1	1000 users results before DB changes . . . . .	52
4.2	1000 users results after DB changes . . . . .	52
5.1	HTTP Post Request for Compilation . . . . .	55
5.2	HTTP Post Request for Execution . . . . .	56
5.3	HTTP Post Request for Evaluation . . . . .	56
5.4	100 users on 10 Apache workers . . . . .	58
5.5	400 users on Engine with server maxconn 400 . . . . .	58
5.6	engine:400 users on Engine with contimeout 50000 . . . . .	59
5.7	engine:400 users on Engine with server maxconn 100 . . . . .	59
5.8	engine:500 users on 50 MaxRequestWorkers . . . . .	60
5.9	500 users on 100 MaxRequestWorkers . . . . .	60
5.10	750 users on 200 MaxRequestWorkers . . . . .	60
5.11	1000 users on 250 MaxRequestWorkers . . . . .	61
6.1	Parameter values for Engine on 8 core machine . . . . .	76

---

6.2	Parameter values for WebApp on 8 core machine . . . . .	76
6.3	Parameter values for Engine on 48 core machine . . . . .	77
6.4	Parameter values for WebApp on 48 core machine . . . . .	77
6.5	Parameter values for Handling Combined Workload on 8 core machine . . .	78
6.6	Parameter values for Combined Workload on 48 core machine . . . . .	78

*Dedicated to my Family.*

# Chapter 1

## Introduction

Programming courses on MOOCs have been found to be heavily favored. A decent course on a popular MOOC can easily have an enrollment of 5000-10000 users. However promising the content may be, the student participation decreases towards the end. It is because there are no time-bounded lab sessions in these courses, where all the students have a sit for a fixed time slot and solve some assignments. Prutor is one such software that was built to conduct programming lab sessions for introductory programming course conducted at IITK. If we aim to integrate it with existing MOOCs, it should be able to handle a load of atleast 5000 active users.

This thesis is all about the scaling Prutor. Now in [1] it is mentioned that by horizontally scaling more servers, they can serve approximately 400 users. We ran the test and found that 8 servers were required to successfully handle the load. Now, the question arises, why cannot we reach such a scale with fewer number of servers? What are the factors, that are being the bottlenecks? Since, Prutor is an in-house developed tool at IIT Kanpur, we assure no such study has been done which answers these questions.

So, in this thesis we first try to hunt the bottlenecks down by profiling [Sec. 3.3] the application. Then we gradually explore the tools used in the technology stack of Prutor and experiment with different values of their configuration parameters. In the process, via benchmarking using standard tools [Sec. 2.2], we have tried to come up with appropriate values for those configuration parameters for different values of user load. We have also experimented with various database optimization techniques. We also removed the database abstraction layer which led to the reduction in the number of queries executed per compilation request. In the end, we present some benchmarking results which compares the performance of Prutor before our changes with the performance after our changes.

We have successfully reached 3000 users on webapp, 1000 users on engine and 1000 users combined.

The next section formalizes our problem.

## 1.1 Problem Statement

To take Prutor from a university scale of 400 active users to a commercial scale of 5000 active users.

## 1.2 Motivation

[1] mentions that the main motivation behind Prutor's development was to abstract out the factors such as programming environments, language-specific build commands, from the process of problem-solving. It addresses those issues head-on and solves them in a very profound manner. It provides a web-based editor interface where a user can compile, execute and evaluate his/her code. Apart from this, the system provides valuable feedback to students based on their submissions, in real-time, but as of now all these facilities are limited to students of a particular university in which it is deployed.

If we integrate a tool like this with the currently popular MOOCs such as coursera [2], Edx [3] and Udacity [4], this will immensely increase the participation among students. It will enable an instructor offering a programming course on these websites to conduct time-bounded exams and labs. From a student's perspective, proctored exams and labs with real-time feedback will keep him/her more engaged.

So handling the load of a course on a MOOC means handling atleast 5000 active users. The capability to just handle 400 active users leaves us very far behind our end goal. Thus a lot needs to be done on the scalability aspect of Prutor. This thesis is the first of the many forthcoming steps in that direction.

## 1.3 Contributions of the thesis

The foremost contribution of this thesis is a scaled up version of Prutor, which can now handle 3000 users on WebApp (Chap. 3) and 1000 users on Engine (Chap. 5) and 1000

users combined. In addition to this, we have demonstrated how we can use standard tools to benchmark Prutor, which had never been done before. The documentation for setup of each tool has been included in the appendix.

We have uncovered what all goes under the hood, when the most frequently accessed APIs of Prutor are hit. We examined these APIs at the query level and experimented with various optimization techniques. We also provide the test plans which were instrumental in the simulation of user behaviour.

We compare the results on 2 machines and show which one performs better. In the end we provide a tabulated summary of the appropriate parameter values of various components in Prutor's technology stack. This summary can act as a guide to Prutor admins for tuning these components before an event.

## 1.4 Organization of the thesis

This chapter gives a brief introduction to the scalability aspect of Prutor and describes our motivation behind picking up this project. It concludes with listing our contributions.

In Chap. 2, we have talked about Prutor from a developer's perspective and briefly explained the tools that we have used for its benchmarking and analysis. It then concludes with the description of the works that were in some sense related to our work.

In Chap. 3 we have described our complete analysis of the WebApp component of Prutor. This chapter tells a story of how we reached 3000 users on WebApp. It also describes some suitable minor code modifications that we did along the way.

Chap. 4 has been dedicated to the Database Analysis. In this chapter we have described our API analysis on the query level. We have also covered tuning of some critical parameters of MySQL server.

Chap. 5 unfolds the story of how we reached 1000 users on Engine. It consists of 2 main subsections. First is about the tuning of various Apache and Proxy parameters. The next subsection describes how we removed the Database Abstraction layer from most heavily used APIs.

In Chap. 6 we have graphically compared the performance of our modifications on 2 machines. It concludes after giving a tabulated summary of appropriate values of some critical parameters of various components of Prutor's technology stack.

---

Chap. 7 concludes our work and describes how our work can be taken forward. In the end we have appendix, which contains the step-by-step installation instructions for various tools that we used. It also contains the links to our version of the Prutor repository along with the test plans that we built for benchmarking.

## Chapter 2

# Background, Tools and Related Work

This chapter briefly describes all the components of Prutor. It then describes the tools that have been used for benchmarking and analysis. It concludes with a related work section which highlights some of the previous works where the same tools were used for benchmarking different applications.

### 2.1 Prutor

Prutor, from the architectural point of view, is completely a dockerized [5] system, i.e. each component of the system is being run inside docker-containers [6]. The further subsections of this chapter will talk about the different components of Prutor. The system primarily consists of 6 components. Each component has been briefly described in further subsections.

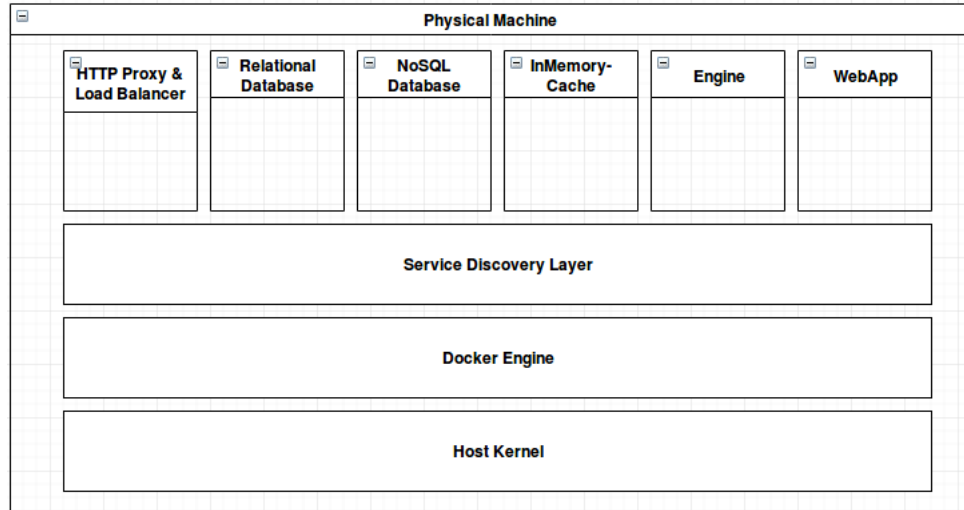


FIGURE 2.1: Prutor: Physical view

### 2.1.1 HTTP Reverse Proxy

**HAProxy** [7] has been used as an L7 Load Balancer [8]. L7 Load Balancer can also be called Reverse Proxy server. It is the first component which the user's HTTP Requests hit. As it works on layer 7, it filters these requests on the basis of the filtration logic written in one of the configuration files of HAProxy.

In Prutor, filtering of HTTP Requests is done on the basis of URL, such that if URL of an HTTP Request is of type `/compile`, `/execute`, `/evaluate` then it is passed to the Engine (Sec. 2.1.6). HTTP Requests related to the remaining functionality are handled by the WebApp (Sec. 2.1.5).

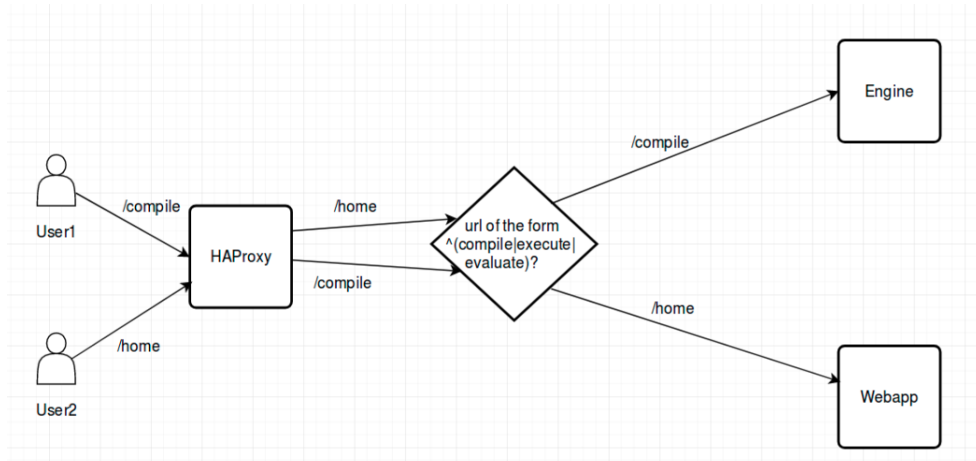


FIGURE 2.2: Role of L7 Reverse Proxy(HAProxy)

### 2.1.2 Relational Database

This component holds all the user data. The user data comprises of their account details, assignments, code submissions, event records and feedbacks to their submitted codes. **MySQL** [9] has been used for this purpose.

### 2.1.3 NoSQL Database

This component holds all the system data and environment settings for the plugins. **MongoDB** [10] has been used for this purpose.

### 2.1.4 In-Memory cache

The in-memory cache component comprises of a server which holds all the data in-memory in the form of key-value pairs. In Prutor, it mainly stores the user-sessions. It has also been used to store the results to some of the database queries. **Memcached** [11] has been used to fulfill this role.

### 2.1.5 WebApp

This is one of the major components of the whole system. It is the back-end server which handles the major portion of HTTP Requests pertaining to the API services provided by Prutor. All the major feature APIs are written on this server. **Node.js** with **Express** [12] has been used for this role.

### 2.1.6 Engine

This component handles the compilation, execution and evaluation requests of the user. It uses Apache to serve the requests. All the code on this component is written in **php** [13]. For each compilation and execution request, it fetches the appropriate(programming environment specific) parameters from **MongoDB** [10] component, compiles/executes/evaluates the code in a sandboxed [14] environment and saves the code and the response in the appropriate tables in MySQL. The response is processed in a readable format and is sent as HTTP response to the user.

## 2.2 Tools used

### 2.2.1 Apache JMeter

Apache JMeter is an open source software which is used for load testing of web applications, databases and servers. It is used to simulate heavy traffic on application servers. For a detailed tutorial on JMeter installation, please refer to the [appendix]. Here we will discuss some of the key jargons that are used in a test plan.

- **Test Plan** - A Test Plan typically describes the steps that JMeter will execute when run. It comprises of a huge number of elements. For a detailed view of the configurations related to a test plan, please visit the Apache JMeter section in Appendix.
- **Thread Group** - The thread group element controls the number of threads, JMeter will generate during a run. The controls for a thread group allow you to:
  - Set the number of users.
  - Set the ramp-up period.
  - Set the number of times to execute the test.
- **Controllers** - JMeter has 2 types of controllers:
  - **Samplers** - Samplers tell JMeter to send requests to a server and wait for a response. They are processed in the order they appear in the tree. Some of the key JMeter samplers include:-
    - \* HTTP Request
    - \* JDBC Request
    - \* FTP RequestEach Sampler has its own options and can be configured to suit user's Requirements.
  - **Logic Controllers** - Through Logic Controllers, Jmeter decide, the order and number of times to send the request. The only Logic controller that I have used is **Only-Once controller**.
    - \* **Only-Once controller**: All the samplers under this controller are only executed once by each thread. It is generally used with HTTP request for login.

- **Listeners** - Listeners provide a tabular or graphical representation of the post-run metrics calculated/collected by JMeter.

The step-by-step instructions have been documented in Chap. [A](#).

### 2.2.2 New Relic APM

New Relic APM is a cloud-based performance monitoring software. It collects real-time performance metrics from your server and presents them in a human readable format in the user's online dashboard. The task of collecting the performance metrics is done by a new relic agent, which is installed on the server or application component, whose performance is to be measured. There are various options in New Relic which really makes the task of finding bottlenecks, little less painful. Some of those key features are:-

- It graphically shows the trend of application's average response time, broken down in intervals of half an hour.
- Color coding of application components in graphs of various performance metrics.
- End to End Profiling of a transaction.
- Ordered representation of transactions on the basis of response time and throughput.
- Database analytics.
- End to End tracing of an transactions.
- Provides APIs for recording custom metrics.
- Error analytics
  - Classification of errors on the basis of error codes.
  - Count of each classification.
  - Mining in the error data and reports some common behaviours among certain types requests.

The setup instructions have been documented in Chap. [C](#).

### 2.2.3 MySQL Workbench

MySQL Workbench is a GUI tool which enables the developers to design, manage and analyse their data in MySQL database. All the key features are available on the press of one mouse click. Following is the list of some important features of MySQL workbench, which make it a delight to use.

- Retrieval of metrics from the performance schema without the need to go into details.
  - Divides the output of performance schema in various classes such as Memory usage, I/O analysis, query analysis and schema analysis.
  - Enables one-click retrieval of important metrics like unused indexes, queries doing full table scans, queries doing on-the-fly sorting and expensive queries, thereby eliminating the need to worry about the underlying schema and query formulation.
  - Eliminates the need of exploring other analysis tools.
- Makes Server administration very easy.
  - Enables viewing Status variables.
  - Enables tuning of System variables.
- Gives a real-time graphical view of the no. of client connections.

Steps to setup are given in Appendix.

The setup instructions can be found in Chap. [B](#).

### 2.2.4 `ps_mem.py`

A python based command line tool to measure RAM used by a process. It calculates private RAM as well as the shared RAM used by a process. It also measures the swap memory that is being used by a process. It has several hooks, which are listed when `h` is given as a hook. Code of this tool can be found at [\[15\]](#).

## 2.3 Related Work

As Prutor is developed in-house, we assure that this kind of work has not been done before for Prutor. However, there are some works which use the approach of benchmarking and performance metrics evaluation in similar context but for other web applications. Vinkal in his MTech Thesis [16] has shown the importance of tuning the appropriate parameters of each component in the software technology stack of the application. His work demonstrates how proper tuning of Apache server can make it handle 180% more users with reasonable response time. Dhananjay in his work [17] has benchmarked and compared the performance of an MMS application built on LAMP and MEAN stacks.

Vipin's work [18] is about analysing the impact of various virtualisation platforms and deployment architectures on the performance of a web application. In this work, the performance assesment has been done via benchmarking with respect to the response time at the client.

Banothu in his work [19] has benchmarked a RESTful web application written in Erlang and Node.js and compared the performance metrics in each case to figure out which performs better. Amit [20] in his work has benchmarked 3 OMG DDS compliant middlewares and compared them on the basis of the performance metrics collected. There has also been a study [21] about the effectiveness of APM tools like New Relic and Dynatrace in detecting performance regressions for web applications.

## Chapter 3

# WebApp: Analysis and Benchmarking

In Prutor, WebApp is the back-end server responsible for majority of the features like user authentication, management of users, problems and events. It is the component which provides the editor interface for programming. All the APIs on this server are written using Express framework in Node.js runtime.

WebApp provides a plethora of features to a user. The accessibility of these features depends on the role of a user. If a user is an admin, he/she can perform administrative tasks like management of users, problems and events. On the contrary, if the user is a student, he/she can only view their assignments and solve them. For benchmarking, we have focussed only on those features which are heavily used in an event. To know about events in Prutor, one should know about perspectives.

### 3.1 Perspectives

Prutor offers 3 perspectives to a student. They are:-

- Scratchpad - This perspective offers an arena, where you can try out your own idea. There is no problem statement. There is no time bound. It just offers you an editor to write code on and interfaces to compile, execute and save your code.
- Practice - This perspective offers an arena, where students can practice old problems. It is same as scratchpad with the difference that here you will write corresponding

to previously asked problems. It offers same functionality as scratchpad with the additional functionality of evaluation of code.

- **Event** - Event is the only perspective in Prutor, which has a time bound. An event can be classified into 3 categories: lab, quiz and exam. The motivation behind creating an event perspective is to simulate an environment which is similar to a proctored examination. Each student in an event has to solve some assignments within the time limit of the respective event. Their submissions are then graded by Teaching Assistants and the process carries on.

The key thing to note here is, Event is the only perspective where the system can get overwhelmed with the amount of incoming user workload. So, if we benchmark our system on the most frequently used features in an event, our job is done. User workload is nothing but HTTP GET/POST requests. The next subsection has been dedicated to these HTTP requests.

### 3.1.1 Student Activities in an Event

In this subsection, I have described the various activities that a student can do in an event. The description highlights the type of HTTP requests involved in carrying out these activities their URL and the parameters they take as input.

<b>Use Case</b>	Login
<b>HTTP Request URL</b>	/accounts/login
<b>Request Type</b>	POST
<b>Parameters</b>	- username - password

TABLE 3.1: User Login

<b>Use Case</b>	View Home Page
<b>HTTP Request Url</b>	/home
<b>Type</b>	GET
<b>Parameters</b>	None

TABLE 3.2: View Home

<b>Use Case</b>	View Codebook
<b>HTTP Request Url</b>	/codebook
<b>Type</b>	GET
<b>Parameters</b>	None

TABLE 3.3: View Codebook

<b>Use Case</b>	Open assignment
<b>HTTP Request Url</b>	/editor/<assignmentId>
<b>Type</b>	GET
<b>Parameters</b>	None

TABLE 3.4: Open assignment

<b>Use Case</b>	Manually Save Code
<b>HTTP Request URL</b>	/editor/save
<b>Request Type</b>	POST
<b>Parameters</b>	<ul style="list-style-type: none"> <li>- assignment_id</li> <li>- branch_id</li> <li>- trigger=manual</li> <li>- code</li> </ul>

TABLE 3.5: Manually save code

<b>Use Case</b>	Submit Code
<b>HTTP Request URL</b>	/editor/save
<b>Request Type</b>	POST
<b>Parameters</b>	<ul style="list-style-type: none"> <li>- assignment_id</li> <li>- branch_id</li> <li>- trigger=submit</li> <li>- code</li> </ul>

TABLE 3.6: Submit Code

## 3.2 Baselineing Scalability of WebApp

Before we go any deeper with the analysis, it is important to know the baseline, i.e. the current capacity of the system. For that we need to perform some performance testing. This section has been dedicated to that.

**Performance testing** comprises of tests that check a system's/application's behaviour on quality attributes such as responsiveness, stability, scalability, reliability, speed and resource usage of your software and infrastructure. Following are some of the performance testing techniques which are relevant to our case.

- **Stress Testing** - Stress testing [22] is adopted to find out the breaking point of SUT(System Under Test) [23]. The testing checks the behaviour of the system under intense loads, and how it recovers when going back to normal usage.
- **Load Testing** - Load testing [22] checks the system's capability to handle heavy concurrent workloads. This type of workload is simulated best by tools [24] like LoadRunner [25], WebLoad [26], Apache JMeter [27].
- **Scalability Testing** - Scalability testing examines the SUT's capability to scale-out. The main goal is to find out the limit beyond which it will resist scaling out.

One can say that the essence of our thesis is Load Testing, but our aim lies in the intersection of load and scalability testing. We have also tried to find out the breaking point of the system, so we have also touched upon stress testing.

### 3.2.1 Distributed Setup

Generation of heavy workloads should not be done on a single machine. For this purpose, we have used Apache JMeter in a **Distributed Master-Slave** setup. The setup comprises of the following components.

- **Master client** This component acts as a controller for the whole setup. It is the master client which sends the test plan to its slave servers at the time of execution. Here Master machine is a physical machine on which Prutor is installed. It has following specifications:-
  - Cores: 8

- RAM: 16GB
- Model: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
- **Slave servers** The slave servers receive the test plan from the master and run it independently of each other. After the job is done, they send the collected metrics to the master, which is then displayed in a tabulated/graphical way to the user. Here the slave server machines are the physical machines, provided by CSE Department of IIT Kanpur. The specifications are as follows.
  - Cores - 4
  - RAM - 8GB
  - Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz

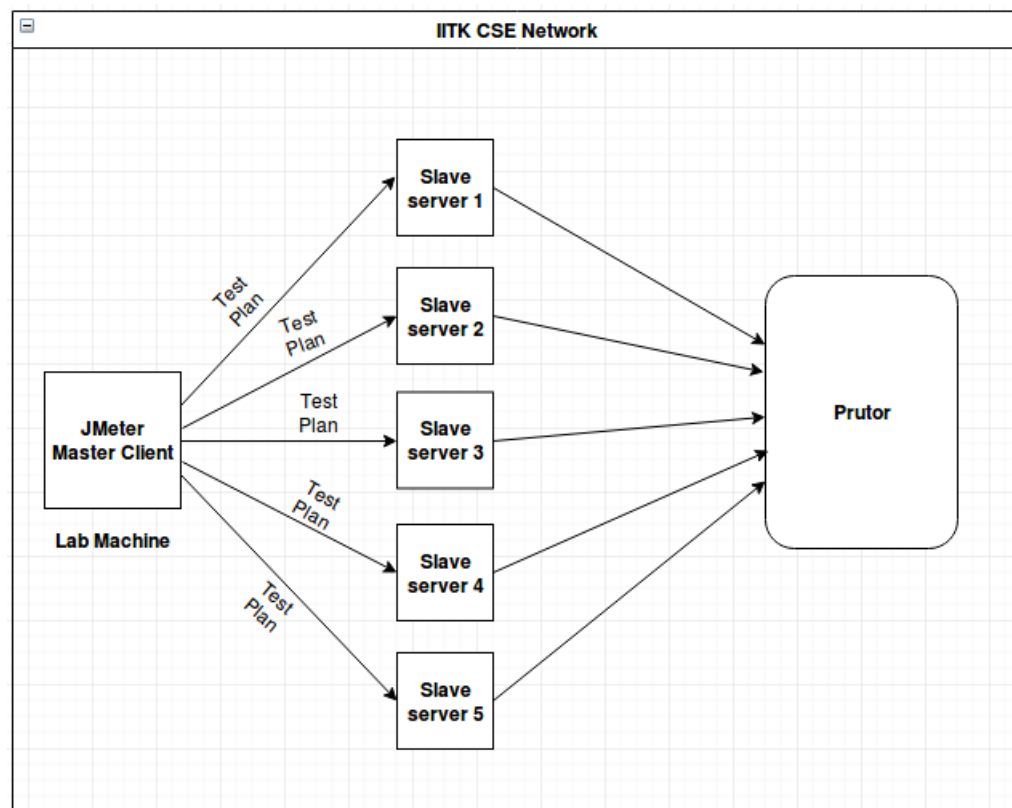


FIGURE 3.1: JMeter:Distributed Master-Slave Setup

The detailed step-by-step instructions to set it up have been documented in Chap. [A](#).

The next subsection provides information about the Workload that was generated by this distributed setup.

### 3.2.2 Workload Specification

The load is generated with the help of Apache JMeter (explained in Sec. 2.2.1). The test plan (explained in Sec. 2.2.1) comprises of some selected HTTP Requests(as mentioned in Sec. 3.1.1) having event perspective. The sequence of the requests in the plan simulates the actual load that a user generates in an event. Some key components in this test plan are:-

- **Virtual Users(VUs)** This setting is controlled by the configuration element **Thread Groups**.
- **Ramp-up Period** This setting controls the time in which JMeter will reach full workload as in equal to the VUs specified. For eg:- if VUs=1000 and Ramp-up period is 10seconds, then it will take 10 seconds to reach the load of 1000 users, or we can say that in each second load of  $100(1000/10)$  VUs will be generated.
- **Iterations** This controls the number of iterations, load of each thread (VU) will be executed.

Apache JMeter has **Constant Timer** which allows you to control the **timing between each request** of a particular VU. This timing has been chosen to be 2 seconds, keeping in mind that the workload does not go too far from the real world scenario. Before moving any further with this value, we first show a comparison of performance between the baseline results without constant timer and with a constant timer of 2 seconds in Sec. 3.2.4.

### 3.2.3 Performance Metrics

We started with finding out the current capacity of the system. In [1], it is mentioned that the system is horizontally scalable and can reach 400 users. So we selected 5 scenarios as 1 web application server (webapp), 2 webapps, 4 webapps, 8 webapps, 16 webapps. For each scenario, we started from 100 VUs and went on till 1000 VUs to find out the load handling capacity of each scenario. For measuring the load handling capacity we are using **Error%** metric of the Login request collected by JMeter. The reason behind choosing the Error% of only login request is that if a VU fails to login, its subsequent requests give a 302 instead of error, thereby resulting in a very less Average Response Time and almost negligible Error% which is completely wrong. Hence we can only trust the Error% of Login requests.

### 3.2.4 Results

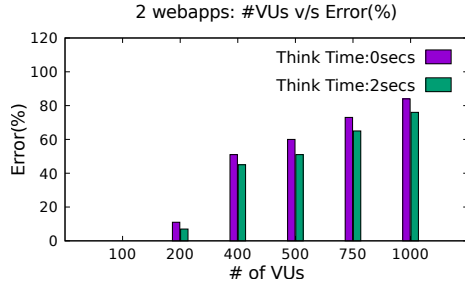


FIGURE 3.2: VUs v/s Error% for 2 webapps

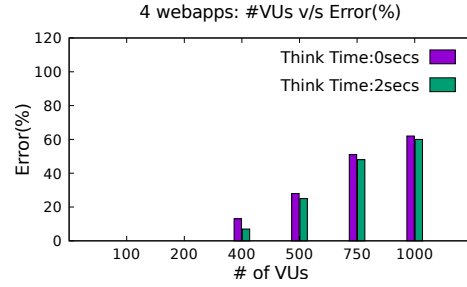


FIGURE 3.3: VUs v/s Error% for 4 webapps

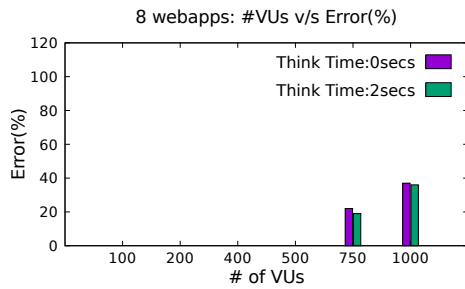


FIGURE 3.4: VUs v/s Error% for 8 webapps

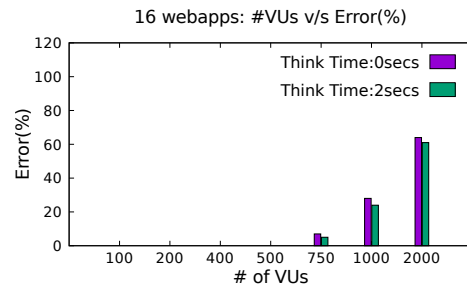


FIGURE 3.5: VUs v/s Error% for 16 webapps

One common observation from the above graphs is that the difference between the green and purple line is very little. This means that even with think time of 2 secs, our system is under a significant amount of stress. Hence we can treat the green line as our baseline and our goal will be to minimize this green line as much as possible. Fig 3.6 shows that for handling the load of 200 users, we need to run 4 webapps, for handling 500 users, 8 webapps are required and with 16 webapps we cannot even reach 750 users. So this section gave us the scalability baseline of Prutor.

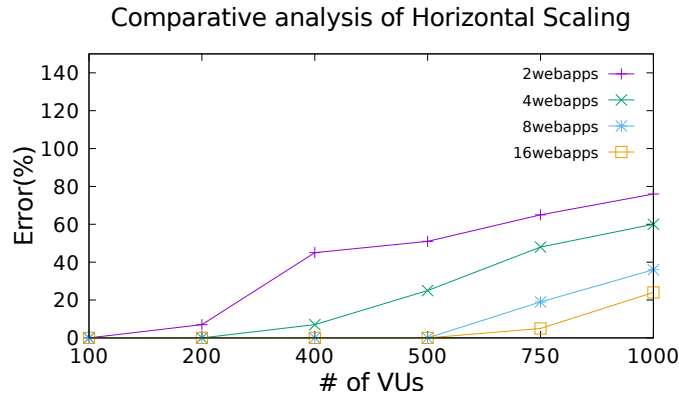


FIGURE 3.6: Comparison of Baseline Error% on different number of webapps

In the next section, we have discussed about performance profiling of Prutor.

### 3.3 Performance Profiling

Performance Profiling [28] is done to record and analyze the performance of each component of an application. The major objective behind performance profiling is identification of the bottlenecks. This section has been divided into subsections containing profiling snapshots of the most frequently accessed features of an event (Sec. 3.1.1).

#### 3.3.1 Setup

Refer to the documentation in Chap. C.

#### 3.3.2 Profiling of each feature

- **View Assignment** - Fig 3.7 shows the list of slowest components hit, when an HTTP GET Request for viewing an assignment is received. Fig 3.7 & 3.8 lists down memcached and readFile as bottleneck. Here `editor:after_db` is a custom tracer provided by New Relic, which we used to localize the readFile component.

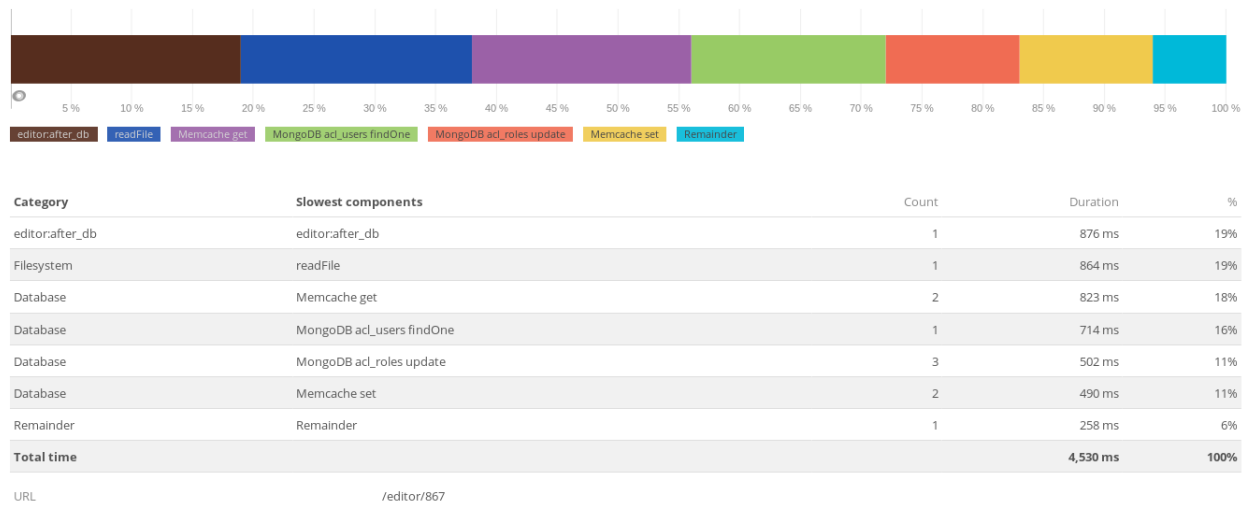


FIGURE 3.7: View Assignments:Slowest Components

Breakdown table

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Custom	editor:after_db	20.1	1.0	39.5
Filesystem	readFile	15.5	1.0	30.6
Database	Memcache.get	13.1	2.0	25.8
Database	MySQL.problem.select	8.2	2.0	16.2
Filesystem	stat	6.4	1.0	12.7
Database	MongoDB.aci_users.findOne	6.3	1.0	12.3
Expressjs	Middleware: <anonymous> /editor/:id	5.7	1.0	11.3
Database	Memcache.set	4.5	2.0	8.84

[Show all segments →](#)

FIGURE 3.8: View Assignments:Breakdown Table

```

router.get('/:id', function(req, res) {
  // TELEMETRY
  /*require('../app_modules/telemetry')
    .logUserActivity(req.session.user.id, 'ASSIGNMENT', 'LAND', null);*/

  var assignmentID = req.params.id;

  var assignments = require('../app_modules/assignments');
  var events = require('../app_modules/events');
  var async = require('async');

  events.isAllowed(req.session.now, assignmentID, function(err, allowed) {
    if(allowed) {
      async.parallel([
        function(callback) {
          assignments.getLastSavedCode(req.session.user.id, assignmentID, callback);
        },
        function(callback) {
          assignments.getAssignmentProblem(assignmentID, callback);
        },
        function(callback) {
          assignments.getAssignmentDetails(assignmentID, callback);
        }
      ], nr.createTracer('editor:after_db', function(err, dataSet) {
        if(err) {
          console.log(err);
          res.send('An error occurred! Could not continue. Please contact the system administrator.');
```

FIGURE 3.9: View Assignments:Custom Tracer

- **View Home Page** - Fig 3.10 & Fig 3.11 both represents readFile and memcached get to be the slowest components, in the control flow when an HTTP GET Request for viewing home page is received.

Breakdown table				
Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Database	Memcache get	80.4	4.0	604
Filesystem	readFile	7.4	2.0	55.7
Expressjs	Expressjs: get /home	2.0	1.0	14.7
Database	MySQL assignment select	1.7	3.0	12.7
Expressjs	Middleware: <anonymous> /home	1.4	1.0	10.7
Database	MySQL notification select	1.1	1.0	7.91
Database	MongoDB ac_l_users findOne	1.1	1.0	7.91
Database	MongoDB ac_l_allows_home	1.1	1.0	7.99

FIGURE 3.10: View Home Page:Breakdown Table

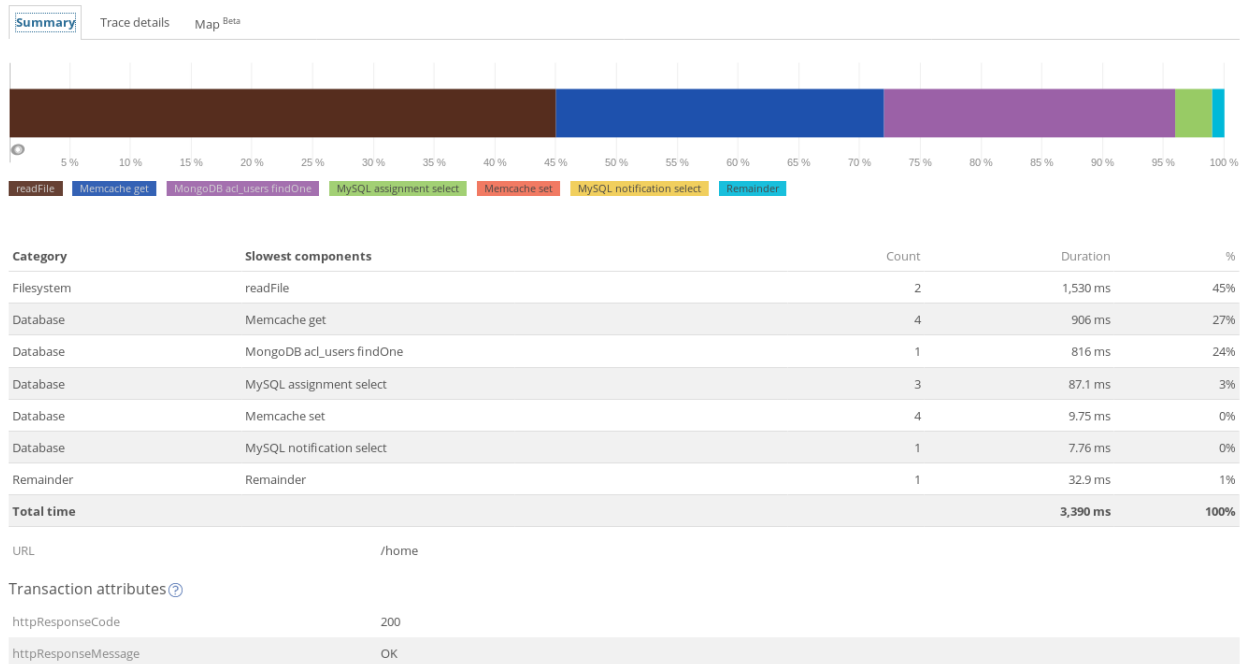


FIGURE 3.11: View Home Page:Slowest Components

- **Login** - Fig 3.12 shows that Express middleware, memcached get/set are the top 3 slowest components in the control flow, when a user logs in.

Breakdown table

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Expressjs	Expressjs: post /accounts/login	28.9	1.0	381
Database	Memcache get	21.3	2.0	281
Database	Memcache set	17.7	2.0	234
Expressjs	Middleware: <anonymous> /accounts/login	14.8	1.0	195
Database	MySQL account select	9.6	2.0	127
Database	MongoDB acl_meta update	6.8	1.0	89.8
Expressjs	Middleware: session /	0.9	1.0	12.3
Expressjs	Middleware: query /	0.0	1.0	0.0025

FIGURE 3.12: User Login:Breakdown Table

- **Save code: Manual and Submission** - Fig 3.13 and 3.14 shows MySQL code insert to be expensive.

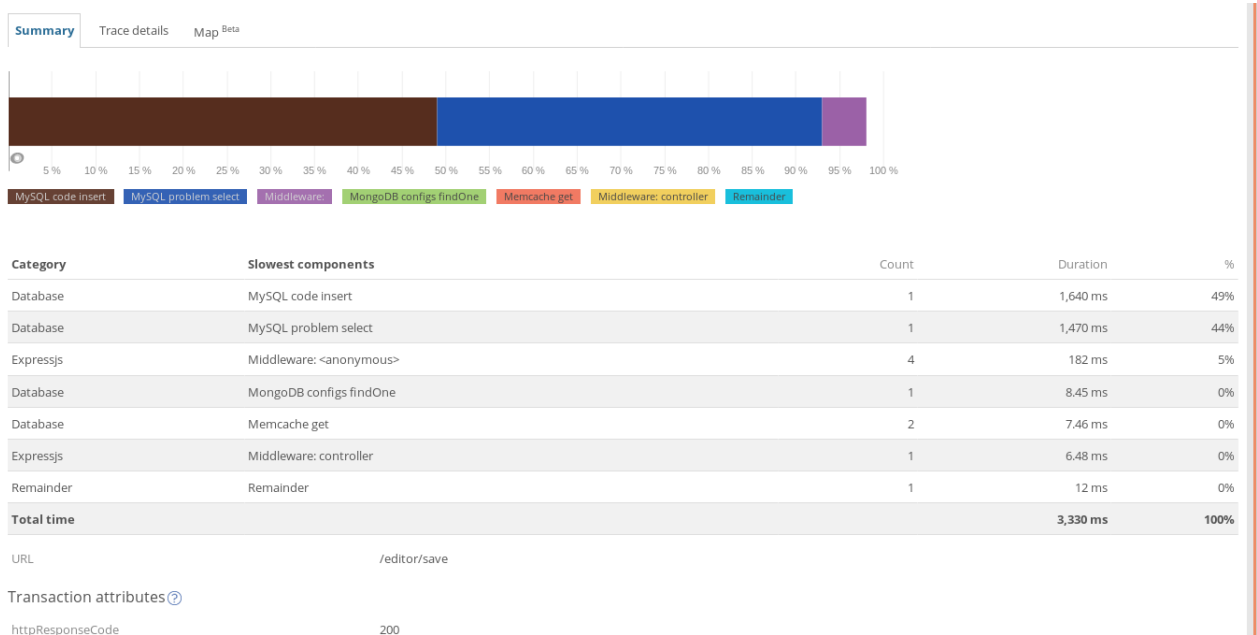


FIGURE 3.13: Save Code:Slowest Components

Breakdown table

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Database	MySQL code insert	57.2	1.0	105
Database	MongoDB configs findOne	15.4	1.0	28.5
Database	Memcache get	10.1	2.0	18.6
Expressjs	Middleware: <anonymous> /editor/save	5.2	1.0	9.56
Database	MySQL problem select	4.5	1.0	8.29
Expressjs	Middleware: controller /	2.1	1.0	3.88
Database	Memcache set	1.7	2.0	3.07
Database	MongoDB acl_users findOne	1.4	1.0	2.59

FIGURE 3.14: Save Code:Breakdown Table

### 3.3.3 Analysis of the common bottlenecks

In this subsection, we have dug further into the common bottlenecks that we found in the previous section.

#### 3.3.3.1 Memcached

In Prutor, Memcached is used for the following use cases.

- Express session-storage.

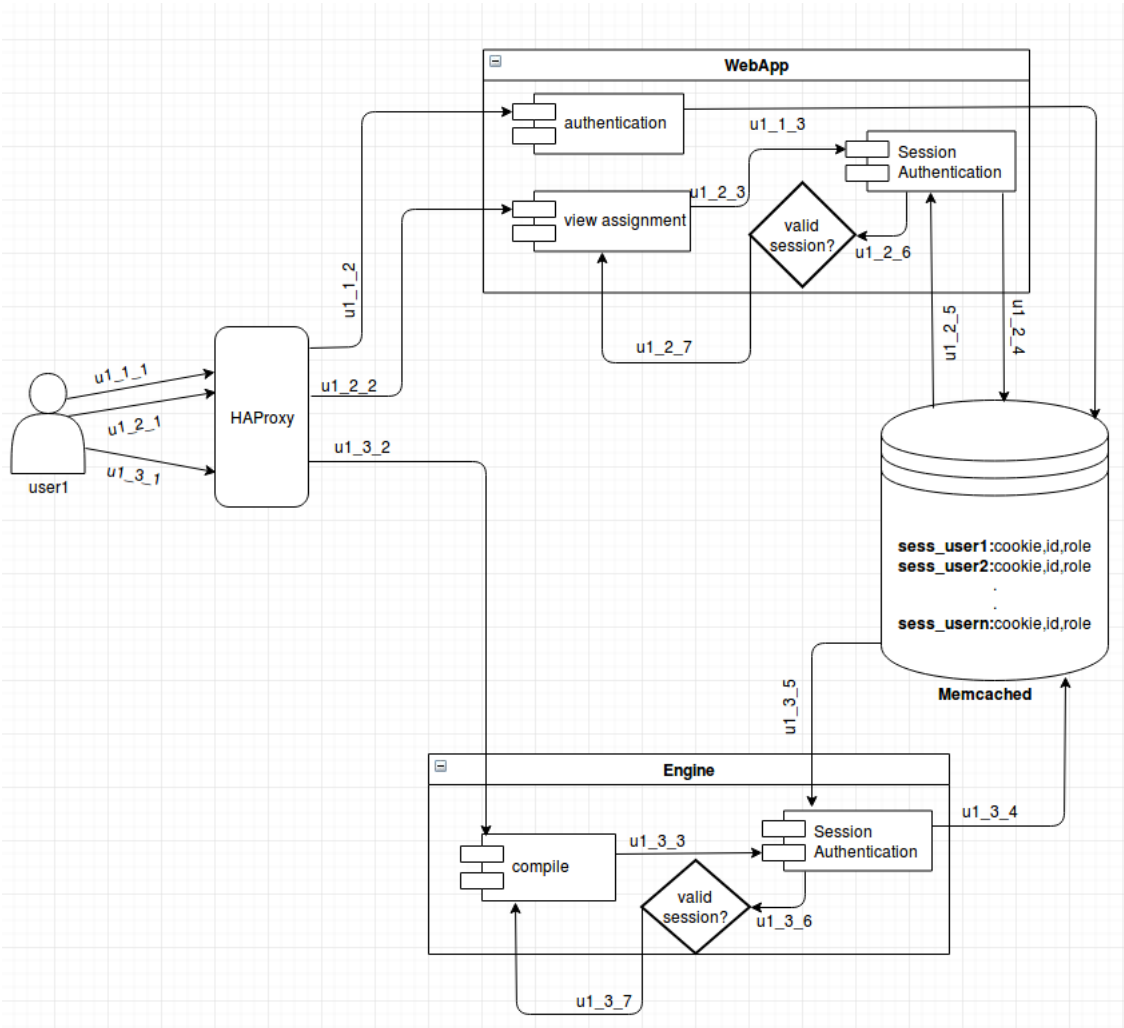


FIGURE 3.15: Memcached:Storage &amp; Retrieval of Sessions

Fig 3.15 describes the use of memcached in user-session storage and retrieval with the help of 3 requests  $u1_1, u1_2, u1_3$ . Here  $u1_i$  represents  $i^{th}$  request by user1 and  $u1_{i,j}$  represents  $j^{th}$  component of the flow of the  $i^{th}$  request.

- **u1\_1** - User sends  $u1_1.1$  or login POST request to HAProxy. The HAProxy then forwards it to WebApp as  $u1_1.2$ . After the authentication is done, a session variable is sent as  $u1_1.3$  to be stored at the memcached server. This session variable stores user's cookies, id and his/her role.
- **u1\_2** - User sends  $u1_2.1$  or view assignment GET request to HAProxy. HAProxy then forwards it to WebApp. At WebApp, it is received by the view assignment module, which then sends it to Session Authentication module. The session authentication module requests the cache server to send session of a particular user

via u1\_2.4. It then matches the user id in the session received(u1\_2.5) and the one received in the parameters. If the session verification is successful(u1\_2.7), then only a user can view his/her assignments.

- **u1\_3** - Similar to above 2 requests, compilation request also sends a get request (u1\_3.4) to the cache server. If available, it then sends that user session variable(u1\_3.5). If authentication is successful, then the compilation is carried out.

- **Cache Layer before Database** - Cache stores data in key-value pairs. An md5 hash of a database query is calculated and is stored as a key. The result to this query is stored as a value in the memcached. So, a query before hitting database, will first hit cache with a get request for the value. If available, it is returned to the user. If not, it fetches the correct value from the database and again hits cache with a set request.

Browsing through the code, revealed that home (Sec. 3.1.1) query heavily used memcached layer before hitting database. So, instead of 5 iterations of the test plan, I executed it for 10 iterations and got the results as follows:-

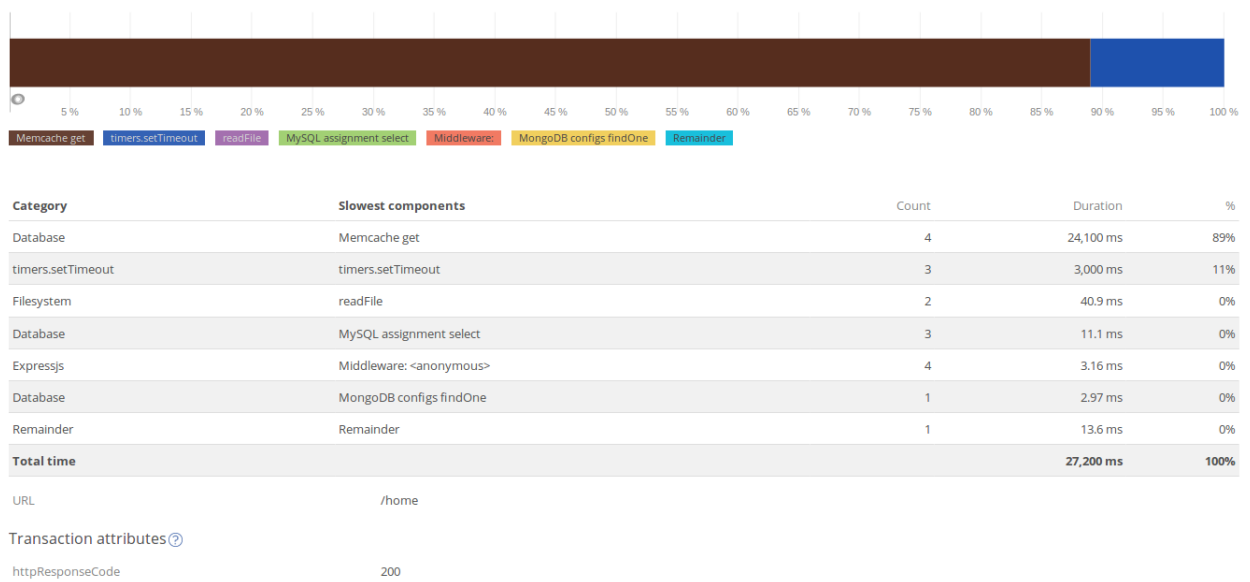


FIGURE 3.16: Memcache: Slowest Components

Duration (ms)	Duration (%)	Segment	Drilldown	Timestamp
9,120	100.00%	get /home		0.000 s
4.65	0.05%	> 14 fast method calls		0.000 s
9,110	99.94%	Router:		0.005 s
9,110	99.94%	Route Path: /home		0.005 s
9,110	99.94%	Middleware: <anonymous>		0.005 s
9,040	99.10%	Memcache get		0.005 s
0.0202	0.00%	net.Socket.connect		0.005 s
1,000	10.97%	timers.setTimeout		5.005 s
0.0521	0.00%	Callback: <anonymous>		9.043 s
9,040	99.10%	Memcache get		0.006 s
0.0122	0.00%	net.Socket.connect		0.006 s
1,000	10.97%	timers.setTimeout		5.005 s
0.071	0.00%	Callback: <anonymous>		9.043 s
3.29	0.04%	MySQL assignment select		9.048 s
2.97	0.03%	MongoDB configs findOne		0.007 s
2.31	0.03%	MySQL event select		0.009 s
2.21	0.02%	MySQL notification select		0.009 s

FIGURE 3.17: Memcache:Detailed Trace

Fig. 3.16 and 3.17 show a lot of time being consumed at connection creation. After browsing through some blogs, I found that by default applications open new connection to memcached server for every get/set request. So while running the test plan for home query for twice the iterations, I kept on doing a netstat on the cache server and the results for 100 users are shown in Fig. 3.18. Here `netstat -avtnp` lists all the TCP connections in which the cache server is involved with. Each entry in this output contains a column for Local Address and Foreign Address. The Local Address is the IP address of the cache server. The Foreign Address is the IP address of the foreign entity that is connected to the cache server. So the output of `netstat -avtnp` has been piped to `grep 172.17.0.9`. The IP 172.17.0.9 was the IP of the webapp container at that time. So now we are filtering only those output entries which involve the webapp container as the foreign entity that is connected to our cache server. Lastly, we are counting those entries by using `wc -l`.

```

root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
0
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
68
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
102
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
88
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
100
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
119
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
146
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
109
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
139
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
147
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
165
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
167
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
167
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
304
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
669
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
985
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
974
root@b990fed6fe9b:/# netstat -avtnp|grep 172.17.0.9|wc -l
1001

```

FIGURE 3.18: Memcache: netstat

- **Possible Fixes**

- Cache Pooling
- Skip the Cache Layer

We will see the effects of these fixes in Sec. [3.4](#).

### 3.3.3.2 readFile

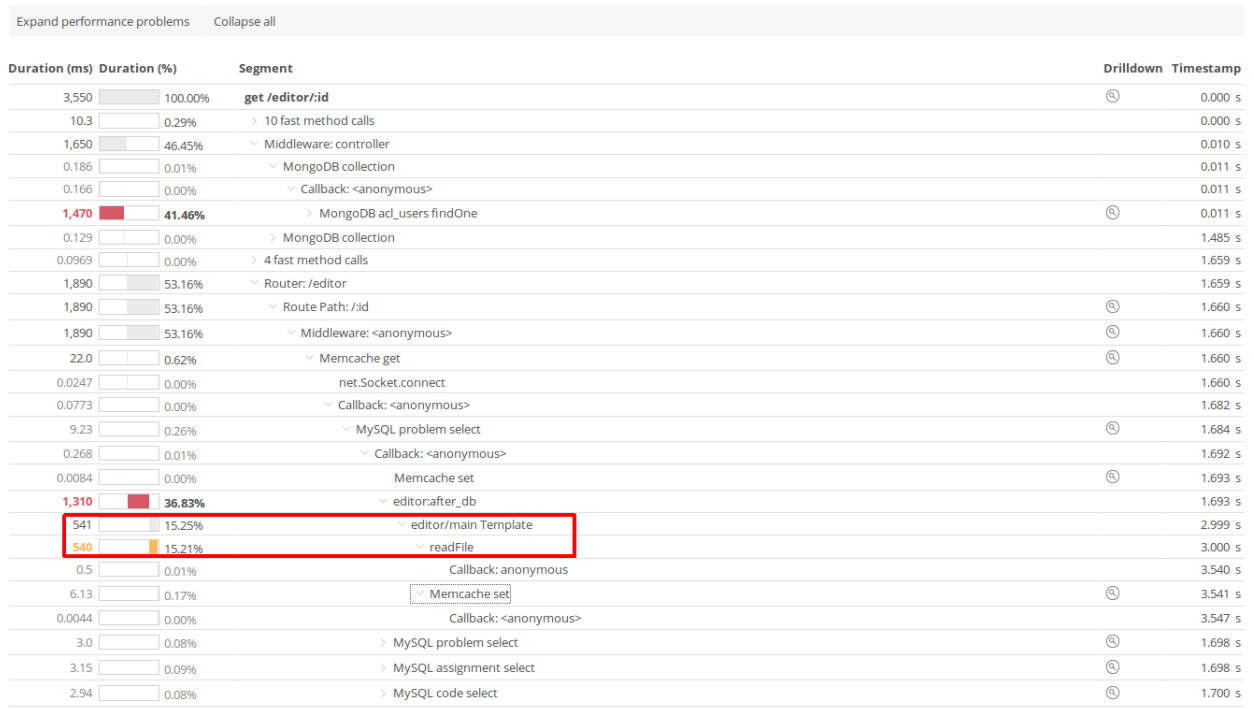


FIGURE 3.19: Slow component: readFile

The `readFile` [29] is a function in Node.js which is used to asynchronously read a file. The reason to it's popping up is that, in order to serve the UI, the Node server renders view template files. For pages consisting of many UI elements, often a large template file is broken down to smaller child template files with the parent file containing placeholders for values to be returned by child template files. At the time of rendering, each of the child template files are read, populated with values and are included in the parent template files. This is the reason for heavy usage of `readFile`.

### 3.3.4 Error Analytics

New Relic provides an error analytics interface, which classifies the errors and extracts some meaningful information from these errors. Fig 3.20 shows the classification of errors and count of each class. As we can see the class with the highest frequency is **Service-UnavailableError**. This error is received either when the server is not able to respond within the timeout period (the time within which a client should receive response) or it

gets too overwhelmed with the incoming HTTP requests, that it starts rejecting them. The error code for this type of error is 503.

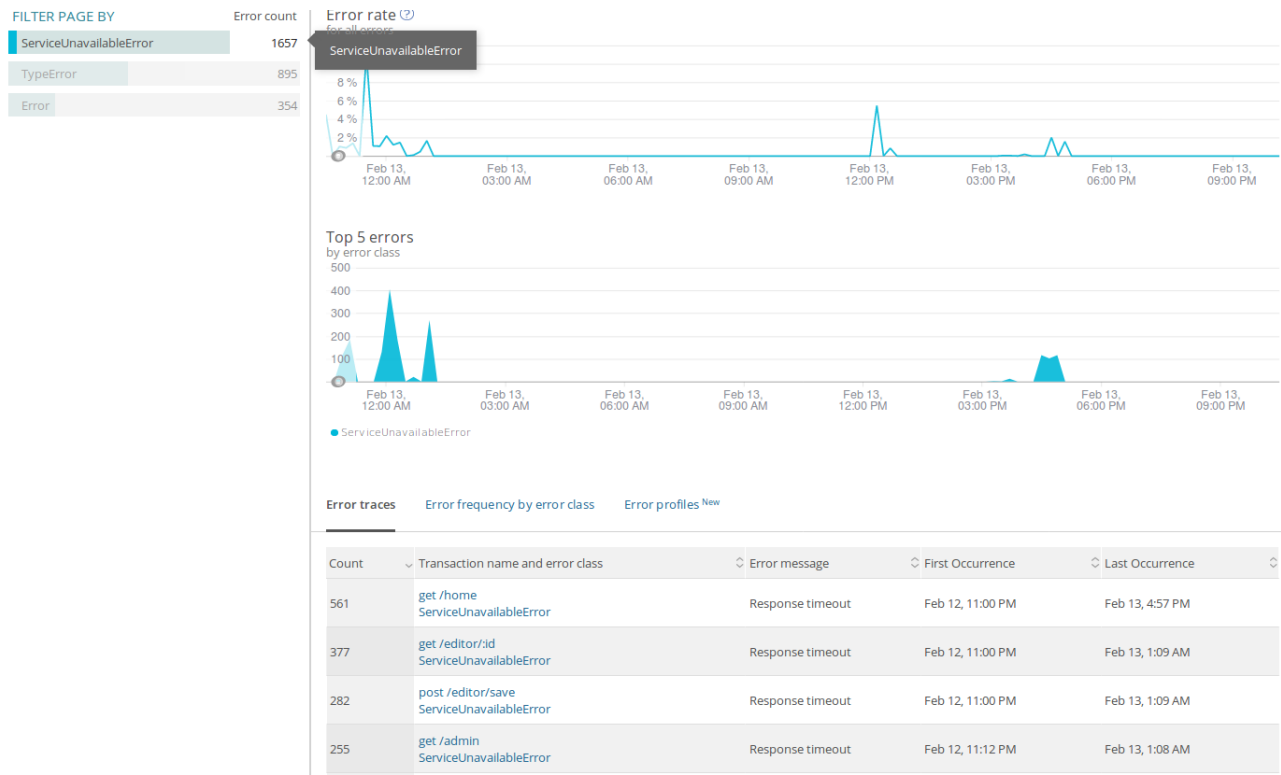


FIGURE 3.20: Error Analytics:Classification

Fig 3.21 and 3.22 show some interesting insights such as 59% errors have http response message Service Unavailable, 57% errors are due to Response Timeout, 59% errors have response code 503, 60% errors have port 3000, 40% errors do not even have database call count.

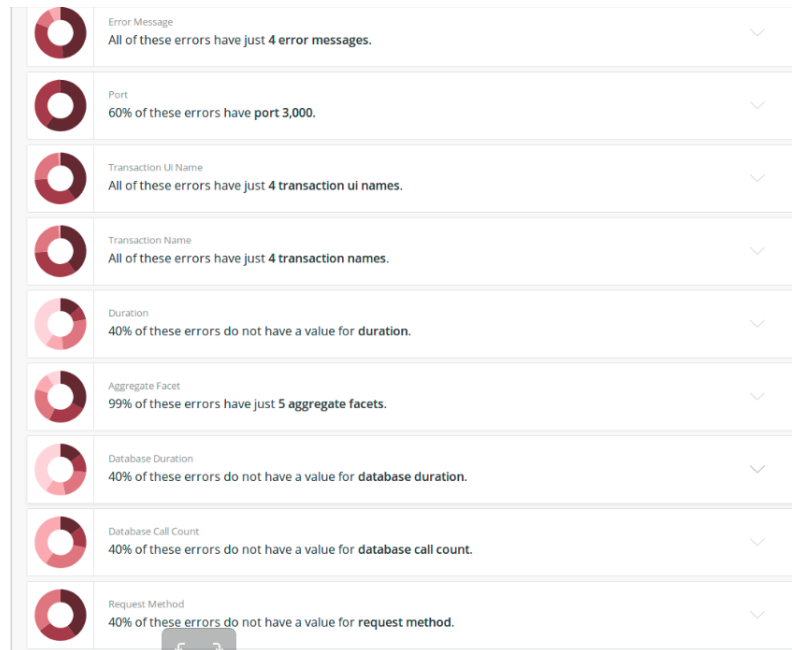


FIGURE 3.21: Error Analytics: part 1

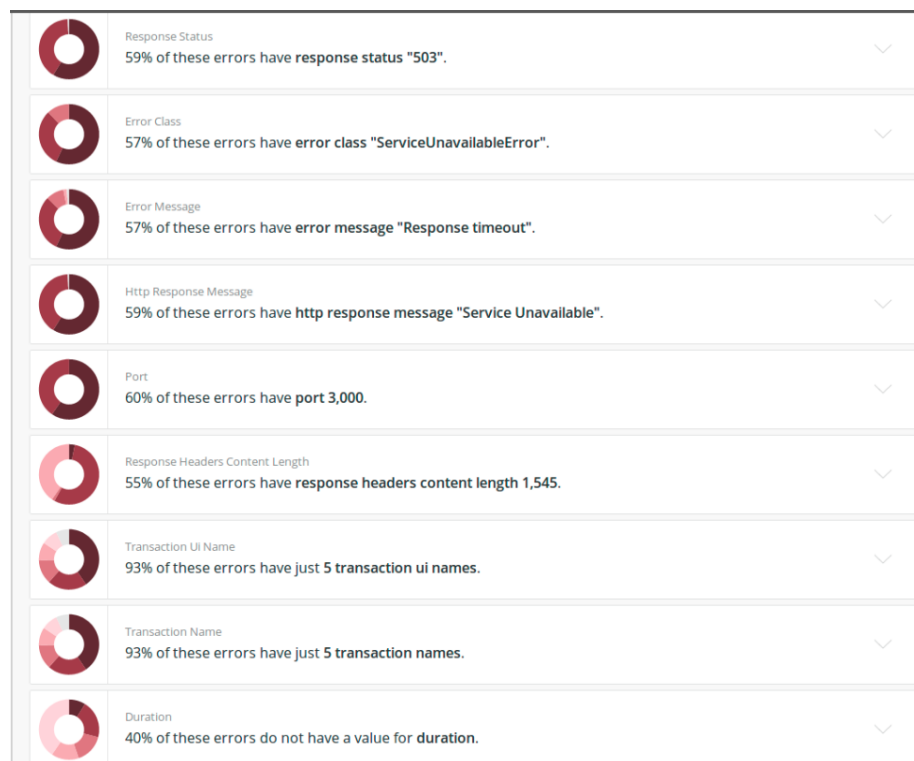


FIGURE 3.22: Error Analytics: part 2

As 40% errors do not even have database call count, so the point of failure is either HAProxy, WebApp or memcached. The Node process in WebApp listens on 3000 port of

the WebApp container. Since 60% of the errors have port as 3000, so we can narrow-down our attention towards HAProxy and WebApp. So we chose HAProxy, since it is the first component in the chain. The next subsection contains the analysis of HAProxy logs and interpretation of its error codes.

### 3.3.4.1 HAProxy logs analysis

Fig 3.23, 3.24 and 3.25 are small segments taken from the logs collected at HAProxy.

```
www web-backend/2e504751ef04 0/0/0/38104/40750 503 1993 - - - - 394/394/365/366/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38098/40744 503 1991 - - - - 394/394/365/366/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38098/40744 503 1991 - - - - 398/398/368/369/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38094/40740 503 1993 - - - - 398/398/369/370/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38094/40739 503 1991 - - - - 397/397/368/369/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38092/40737 503 1995 - - - - 397/397/369/370/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38087/40731 503 1995 - - - - 397/397/369/370/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38087/40732 503 1993 - - - - 406/406/371/372/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38084/40728 503 1991 - - - - 401/401/370/371/0 0/0 "POST /accounts/login
www web-backend/2e504751ef04 0/0/0/38082/40727 503 1997 - - - - 401/401/370/371/0 0/0 "POST /accounts/login
```

FIGURE 3.23: HAproxy:503

```
www web-backend/<NOSRV> 4/5000/-1/-1/5004 503 213 - - sQ-- 99/99/99/0/0 0/21 "GET /home HTTP/1.1"
www web-backend/<NOSRV> 0/5000/-1/-1/5000 503 213 - - sQ-- 99/99/99/0/0 0/23 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5000/-1/-1/5000 503 213 - - sQ-- 98/98/98/0/0 0/22 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5001/-1/-1/5001 503 213 - - sQ-- 99/99/99/0/0 0/24 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5000/-1/-1/5000 503 213 - - sQ-- 99/99/98/0/0 0/25 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5002/-1/-1/5002 503 213 - - sQ-- 99/99/99/0/0 0/26 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5000/-1/-1/5000 503 213 - - sQ-- 99/99/99/0/0 0/27 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5000/-1/-1/5000 503 213 - - sQ-- 99/99/98/0/0 0/28 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5001/-1/-1/5001 503 213 - - sQ-- 99/99/99/0/0 0/29 "POST /accounts/login HTTP/1.1"
www web-backend/<NOSRV> 0/5000/-1/-1/5000 503 213 - - sQ-- 99/99/98/0/0 0/30 "POST /accounts/login HTTP/1.1"
```

FIGURE 3.24: HAproxy:sQ

```
www web-backend/2e504751ef04 0/0/0/28566/78566 503 1874 - - sD-- 399/399/399/399/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28566/78565 503 1874 - - sD-- 398/398/398/398/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28565/78565 503 1874 - - sD-- 397/397/397/397/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28566/78565 503 1874 - - sD-- 396/396/396/396/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28566/78567 503 1874 - - sD-- 395/395/395/395/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28567/78567 503 1874 - - sD-- 394/394/394/394/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28567/78567 503 1874 - - sD-- 393/393/393/393/0 0/0 "GET /home HTTP/1.1"
www web-backend/2e504751ef04 0/0/0/28567/78567 503 1874 - - sD-- 392/392/392/392/0 0/0 "GET /home HTTP/1.1"
```

FIGURE 3.25: HAproxy:sD

As we can see that all these logs have 1 thing in common and that is the error code. They all have error code 503, but the reason of 503 is different. All these error codes have special meanings and indications as follows.

- **sQ** - It occurs when the client session has spent a lot of time waiting for a connection slot with the back-end server and timeout period of queue has expired. It also happens when the timeout period for a connection attempt to succeed has also expired.
- **sD** - It occurs when the connection with the back-end server has been established, but before server could respond back to proxy, server timeout struck.

### 3.4 Incremental Modifications

This section tells a story of how we reached 3000 users on a single machine. We started with the direction provided by the analysis results of Sec. 3.3.4. We then tried to follow the analysis results of profiling. This was then followed by some minor code flow optimizations.

This section has been divided into 3 subsections : reaching 500 users, reaching 1000 users and finally 2000 users. Each subsection describes how we decided when to tune a parameter, which parameter to tune, when to increase the number of servers. Those parameters have been briefly described below.

- Node Workers - It is the number of Node.js server instances.
- server maxconn - It is a parameter of HAProxy. Each front-end and back-end servers have their own server maxconn. It limits the number of concurrent client connections that Proxy will pass to the appropriate server.
- listen maxconn - It limits the number of connections per HAProxy listener.

#### 3.4.1 Reaching 500 on 2

- **100 users**

On firing a load of **100 VUs on 1 webapp** we got the following results. From now on instead of webapp, we will use the term Node Workers. Here Node Workers refers to the Node.js server process.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	100	3970	6007	6402	11.000%	2.06992
home	1500	1033	2119	6093	0.533%	17.62073
codebook	500	937	1490	6238	1.600%	6.14394
open_assignment	1000	734	1397	4330	0.300%	12.81657
save_code	2000	751	1469	2399	0.000%	27.37439
TOTAL	5100	912	1546	5571	0.588%	59.30853

TABLE 3.7: 100 requests on 1 Node worker with no tuning

We see an Error of 11%. We then moved our focus to the images in Sec. 3.3.4, specially 3.24 and 3.23. The cause of these error codes and there possible fixes have

been well explained in Sec. 3.3.4.1. So we preferred increasing the **server maxconn** parameter from **20 to 50** and also increased the **listen maxconn** parameter to **100** as listen maxconn should always be greater than server maxconn. The results are as follows.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	100	4286	7220	7720	0.000%	1.99732
home	1500	393	166	8900	0.000%	9.90655
codebook	500	199	140	4856	0.000%	3.96175
open_assignment	1000	60	115	203	0.000%	7.74425
save_code	2000	112	175	322	0.000%	15.22789
TOTAL	5100	275	173	6959	0.000%	33.07715

TABLE 3.8: 100 requests on 1 Node Worker with server maxconn tuned

- **Following Profiling analysis** We tried to follow the analysis result in Sec. 3.3.3, but the results were not what we expected.

– **Pooling of connections**

We tried with cache pooling, but got no significant results. To implement cache pooling, we just added the parameter poolSize in memcached options [11]. This worsened the result because the requests kept on waiting for connections. We tried tuning the parameter, but it did not workout, so ultimately we ended up dropping the idea of pooling of connections.

- **Skipping the Cache layer** There are 3 modes to get a connection to query MySQL.

- \* **sqlquery** A new connection is created before each request.
- \* **sqlpool** A connection is borrowed from a pool of already established persistent connections.
- \* **sqlcache** This is the mode where cache layer kicks in. A hash of the query is prepared and then that hash value is queried within memcached. If a result is available, it is returned as a response, otherwise it uses a pool connection to fetch the required value from the database.

So we tested the same load without the cache layer to see whether there is any performance drop and here are the results. We achieved this by finding out the queries which were being done in the sqlcache mode, and replaced their mode with sqlpool. Table. 3.9 shows the results. As we can see that, there

is no performance drop, we ended up converting all the sqlcache mode queries to sqlpool. When we say we removed the cache layer, we do not include the session retrieval from memcached. That cannot be touched.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	100	4876	7248	7938	0.000%	1.98697
home	1500	381	171	8853	0.000%	9.92786
codebook	500	162	154	4178	0.000%	4.02963
open_assignment	1000	65	147	190	0.000%	7.78416
save_code	2000	106	141	368	0.000%	15.33119
TOTAL	5100	278	169	6965	0.000%	33.13280

TABLE 3.9: 100 users 1 Node Worker with Cache removed

- **200 users** Till now we have tuned **server maxconn** to 50 and **listen** connection limit of HAProxy to 100. On increasing the users to 200, we received **47%** failure rate. To fix this we increased the listen maxconn limit to **500** which brought down the error rate to **43%**. The HAProxy logs showed 2 kinds of logs. Majority of the login queries were having sQ error code and other queries had simple 503. To remove sQ, we increased the server maxconn to **200** which brought down the error% to **1.5%**. Now the logs did not contain any error code. **So it was time we increased the number of webapps.**

- **Node.js Cluster**

Node.js runs on a single process single thread of execution. Thus with a single node process we cannot utilize different cores of the machine. This leads to inefficient usage of the resources. **Cluster** module [30] enables the creation of several node processes, **which can share a server port**. The **parent** process is responsible for the creation of **worker** processes. These worker processes will handle the application logic which we want to parallelize. Apart from offering concurrency, the cluster module offers following advantages.

- Internal load balancing of incoming requests among worker processes.
- In our application, if we want to increase the number of webapps, we can do it either by running more containers or by spawning more worker processes. The latter one comparatively consumes less memory and is hence lightweight.

We referred the code at [31].

So considering the logs and the error% of **1.15%** in the case of 200 VUs, we decided to increase the number of webapps to **2**. This time we did it by increasing the number of worker processes to 2. This perfectly solved our problem. Here are the results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	200	3905	6846	7559	0.000%	3.94571
home	3000	333	199	6911	0.000%	19.79231
codebook	1000	362	68	7168	0.000%	7.73335
open_assignment	2000	32	53	219	0.000%	15.26834
save_code	4000	119	206	677	0.000%	30.25101
TOTAL	10200	263	202	6386	0.000%	66.19036

TABLE 3.10: 200 users 2 workers on WebApp

So at this point, the # node workers were **2**, server maxconn was **200** and listen maxconn was **500**.

- **400 users** On executing the test plan for 400 VUs under the above mentioned conditions, we got **20%** error%. The logs had sQ error code, so increased server maxconn to **400** and listen maxconn to **1000**, which worked.
- **Login Optimization** The Login method needed 2 database queries. First request was to fetch the user type i.e. student or admin. Based on the result of the first query, a function specific to the user type was being called, which used to perform the authentication. Both the queries involved same table, so we merged the attributes of both the queries and formed a single query to perform the login task.

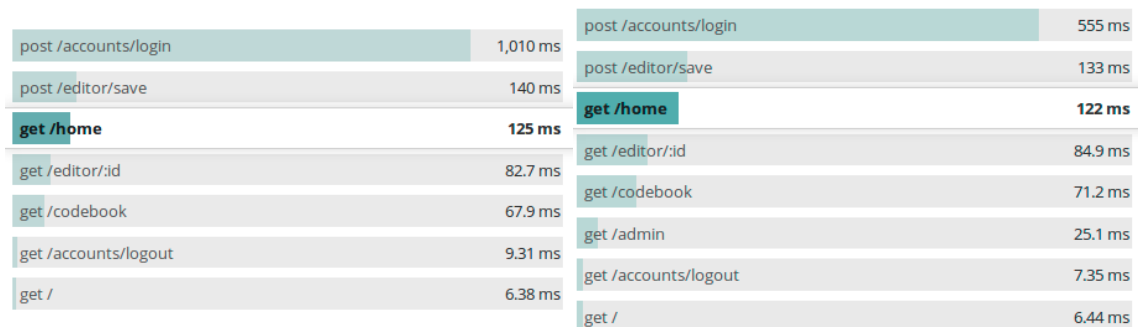


FIGURE 3.26: Average Response Times before Login Optimization

FIGURE 3.27: Average Response Times after Login Optimization

Although the `get/home` query is highlighted, but compare the Average Response Time of `post/accounts/login` in Fig 3.26 and 3.27.

Apart from these changes, there were few redirects after successful login which were removed.

The login request takes the longest to respond, because it involves authentication. The authentication involves usage of `compareSync` function of the `bcrypt` module of npm. The name suggests that this is a synchronous function. This was replaced with it's asynchronous counterpart `compare`.

```
2018-03-14T07:41:28.623Z - info: bcrypt:174ms
2018-03-14T07:41:28.643Z - info: bcrypt:173ms
2018-03-14T07:41:28.678Z - info: bcrypt:171ms
2018-03-14T07:41:28.687Z - info: bcrypt:172ms
2018-03-14T07:41:28.804Z - info: bcrypt:173ms
2018-03-14T07:41:28.821Z - info: bcrypt:174ms
2018-03-14T07:41:28.855Z - info: bcrypt:173ms
2018-03-14T07:41:28.865Z - info: bcrypt:173ms
2018-03-14T07:41:28.993Z - info: bcrypt:173ms
2018-03-14T07:41:29.000Z - info: bcrypt:174ms
2018-03-14T07:41:29.032Z - info: bcrypt:172ms
2018-03-14T07:41:29.043Z - info: bcrypt:172ms
```

FIGURE 3.28: Time taken by Bcrypt

In Fig 3.28, the time is calculated by subtracting the time calculated before and the usage of `bcrypt` module. It takes no less than **170ms** for each login request.

- **500 users** After the above mentioned modifications, we simply tested it on 500 users and got 0.16% error%. The logs had `sQ` error codes. Therefore we again increased the server `maxconn` to 500 and everything worked.

### 3.4.2 Reaching 1000 on 4

In the previous section, we saw that for 500 users, we need 2 node workers, server `maxconn` of 500 and listen `maxconn` of 1000. In this section we have described how we reached 1000 users on 4 node workers.

- **750 users** Without changing any parameter we executed the load of 750 users on the system and here are the results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	750	18176	37156	40626	20.533%	9.07595
home	11250	1084	288	32835	15.404%	60.57408
codebook	3750	820	267	26049	22.160%	23.94132
open_assignment	7500	247	151	5003	20.307%	47.08512
save_code	15000	215	355	5002	16.220%	95.85461
TOTAL	38250	889	365	28690	17.448%	203.26607

TABLE 3.11: 750 users 2 webapps server maxconn 500

On increasing server maxconn to 1000 and listen maxconn to 5000, the error rate came down to **4.67%**. Here is the table.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	750	26243	47193	50004	4.667%	7.99821
home	11250	1889	248	45029	0.347%	56.88800
codebook	3750	585	45	26986	0.000%	22.86892
open_assignment	7500	60	121	260	0.000%	58.05223
save_code	15000	100	174	322	0.000%	114.08406
TOTAL	38250	1179	184	39848	0.193%	191.04749

TABLE 3.12: 750 users 2 workers server maxconn 1000

This time, error logs contained the codes sH and SC. Here are the descriptions of these error codes.

- **sH** - It is similar to sD in Sec. [3.3.4.1](#).
- **SC** - It occurs when the server explicitly refuses to accept the connection request from HAProxy. One fix is to increase the number of servers. The other fix is to tune the network stack of the server.

We picked SC error code first. To tune the network stack we followed [\[32\]](#), [\[33\]](#) and [\[34\]](#). All of them suggested tuning of 3 parameters majorly. They are as follows.

- **net.core.somaxconn** - Controls the number of sockets that kernel can open simultaneously.
- **net.ipv4.tcp\_max\_syn\_backlog** - Controls the number of half open connections(for which ACK has not been received) can be kept in the queue.

- **net.ipv4.tcp\_fin\_timeout** - The `fin_timeout` defines the minimum time these sockets will stay in `TIME_WAIT` state (unusable after being used once).

Unfortunately tuning these values did not work out for us. So we kept them at their default values.

A possible solution to fix sH error code was to increase the **srvtimeout** parameter, but in the table. 3.12 if we look at the 99% line we see **50004** as the maximum value, but the `srvtimeout` was originally set to **50000** seconds. So increasing the timeout further did not seem an appropriate thing to do.

So we increased the node workers from **2** to **4** and it worked.

- **1000 users** Executing the test for 1000 users on the same conditions gave the following result.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	1000	16388	29310	32249	0.000%	13.10170
home	15000	1212	248	28135	0.000%	83.76940
codebook	5000	540	67	20828	0.000%	33.05261
open_assignment	10000	60	146	269	0.000%	76.51344
save_code	20000	144	272	735	0.000%	150.54686
TOTAL	51000	799	249	25022	0.000%	281.13889

TABLE 3.13: 1000 users 4 Node workers

### 3.4.3 Reaching 2000 on 8

- **1500 users** On executing the test for 1500 users in the same conditions, we got the following result.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	1500	18185	36104	36867	31.467%	19.31098
home	22500	908	335	27863	15.591%	123.40939
codebook	7500	607	333	19015	22.253%	49.19678
open_assignment	15000	228	221	5002	20.200%	96.38802
save_code	30000	196	361	1876	15.527%	195.21972
TOTAL	76500	805	374	26302	17.434%	414.30181

TABLE 3.14: 1500 users on 4 workers

As we can see in the table 3.14 that all the requests have error%. This shows that the load of 1500 users is suffocating the 4 node workers, but before increasing the workers, we increased the server maxconn parameter to **1500** which gave the following results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	1500	26227	44888	45658	14.733%	17.02302
home	22500	1510	503	37963	12.693%	113.89637
codebook	7500	454	455	16152	20.000%	46.33434
open_assignment	15000	187	376	493	19.947%	109.19494
save_code	30000	207	469	1027	13.427%	214.50175
TOTAL	76500	1121	480	36262	15.159%	382.30502

TABLE 3.15: 1500 users on 4 workers server maxconn 1500

So still there is some service outage. Now was the time to increase the number of node workers. So from now the number of node workers will be **8**. Now instead of running the test for 1500 users, we executed the test for 2000 users.

- **2000** Uptil now, we have server maxconn at 1500, listen maxconn at 5000 and there are 8 node workers running. For 2000 users here are the results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	2000	17442	32850	35989	8.250%	25.05010
home	30000	1775	1251	31774	0.983%	136.72347
codebook	10000	1323	1152	23972	1.990%	53.15389
open_assignment	20000	853	1299	2850	0.610%	103.95875
save_code	40000	672	1174	2143	0.250%	211.20216
TOTAL	102000	1425	1301	27389	0.864%	459.94427

TABLE 3.16: 2000 users on 8 workers

Increased the server maxconn to **2000** and got the following results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	2000	19686	35321	38799	0.000%	24.10016
home	30000	1977	1166	33934	10.150%	134.98252
codebook	10000	1061	978	23871	19.020%	52.08360
open_assignment	20000	632	1068	1273	20.000%	119.81716
save_code	40000	512	1023	1452	15.745%	236.66118
TOTAL	102000	1396	1114	30067	14.946%	454.14676

TABLE 3.17: 2000 users on 8 workers server maxconn 2000

Now, these results were quite strange. In all the above results, it never happened that all the requests had error% except for login. Since, the login request is a part of Once-only controller and is sent towards the beginning, we speculated that other requests due to their quantity have exhausted a component towards the end. Now proxy did not seem to be an appropriate option, because if it was the reason, we should have seen similar results atleast once before. Memcached was removed earlier in section 3.4.1 from other requests. Now the only heavily used component left was MySQL database.

In section 3.2.4 we saw that after 4 webapps we had to increase the **max\_connections** parameter to **500**. When we looked for the active processes on MySQL, we found it to be 501. The connection limit of MySQL is **max\_connection+1**. Here is the snapshot taken at that time.

```

31015 | prutor | 172.17.0.9:49324 | Its | Query | 0 | query end | UPDATE assignment SET is_submitted=1,submission=1107019 WHERE is_deleted=0 AND id='1014'
31016 | prutor | 172.17.0.9:49344 | Its | Query | 0 | query end | INSERT INTO code (user_id,assignment_id,contents,save_type) VALUES ('6d967cc3-fdb9-4845-8fed-f62e605
31017 | prutor | 172.17.0.9:49454 | Its | Query | 0 | query end | UPDATE assignment SET is_submitted=1,submission=1107060 WHERE is_deleted=0 AND id='45758'
31018 | prutor | 172.17.0.9:49456 | Its | Sleep | 0 | | NULL
31019 | prutor | 172.17.0.9:49468 | Its | Query | 0 | query end | INSERT INTO code (user_id,assignment_id,contents,save_type) VALUES ('b3ed538f-2135-4275-9f2f-7b6a3c8
31020 | prutor | 172.17.0.9:49470 | Its | Query | 0 | query end | UPDATE assignment SET is_submitted=1,submission=1107036 WHERE is_deleted=0 AND id='25283'
31021 | prutor | 172.17.0.9:49472 | Its | Sleep | 0 | | NULL
31022 | prutor | 172.17.0.9:49474 | Its | Sleep | 0 | | NULL
31023 | prutor | 172.17.0.9:49476 | Its | Sleep | 0 | | NULL
31024 | prutor | 172.17.0.9:49478 | Its | Sleep | 0 | | NULL
31025 | prutor | 172.17.0.9:49496 | Its | Sleep | 0 | | NULL
31026 | prutor | 172.17.0.9:49502 | Its | Sleep | 0 | | NULL
31027 | prutor | 172.17.0.9:49522 | Its | Sleep | 0 | | NULL
31028 | prutor | 172.17.0.9:49556 | Its | Sleep | 0 | | NULL
31029 | prutor | 172.17.0.9:49558 | Its | Sleep | 0 | | NULL
31030 | prutor | 172.17.0.9:49562 | Its | Sleep | 0 | | NULL
31031 | prutor | 172.17.0.9:49566 | Its | Sleep | 0 | | NULL
31032 | prutor | 172.17.0.9:49568 | Its | Sleep | 0 | | NULL
31033 | prutor | 172.17.0.9:49570 | Its | Sleep | 0 | | NULL
31034 | prutor | 172.17.0.9:49572 | Its | Query | 0 | query end | UPDATE assignment SET is_submitted=1,submission=1107015 WHERE is_deleted=0 AND id='62439'
31035 | prutor | 172.17.0.9:49574 | Its | Sleep | 0 | | NULL
31036 | prutor | 172.17.0.9:49586 | Its | Sleep | 0 | | NULL
31037 | prutor | 172.17.0.9:49600 | Its | Query | 0 | query end | UPDATE assignment SET is_submitted=1,submission=1107062 WHERE is_deleted=0 AND id='25584'
31038 | prutor | 172.17.0.9:49602 | Its | Sleep | 0 | | NULL
31039 | prutor | 172.17.0.9:49608 | Its | Sleep | 0 | | NULL
31040 | prutor | 172.17.0.9:49610 | Its | Sleep | 0 | | NULL
31041 | prutor | 172.17.0.9:49620 | Its | Sleep | 0 | | NULL
31042 | prutor | 172.17.0.9:49622 | Its | Query | 0 | query end | UPDATE assignment SET is_submitted=1,submission=1107077 WHERE is_deleted=0 AND id='17394'
31043 | prutor | 172.17.0.9:49626 | Its | Sleep | 0 | | NULL
31044 | prutor | 172.17.0.9:49634 | Its | Sleep | 0 | | NULL
31045 | prutor | 172.17.0.9:49638 | Its | Sleep | 0 | | NULL
31046 | prutor | 172.17.0.9:49640 | Its | Sleep | 0 | | NULL
31059 | prutor | 172.17.0.9:49924 | Its | Sleep | 0 | | NULL
31196 | root | localhost | NULL | Query | 0 | NULL | show processlist
501 rows in set (0.00 sec)
mysql>

```

FIGURE 3.29: MySQL:500 connections exhausted

After this we increased the max\_connection limit from 500 to **1000** and got these results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	2000	20082	35746	39235	0.000%	23.95841
home	30000	1843	885	34301	0.000%	130.33958
codebook	10000	977	759	22866	0.010%	49.83380
open_assignment	20000	813	1333	3529	0.370%	114.69534
save_code	40000	534	866	2489	0.278%	227.75541
TOTAL	102000	1401	971	30465	0.182%	438.86635

TABLE 3.18: 2000 users on 8 Node workers

### 3.4.4 Reaching 3000 on 16

On 16 node worker processes, for load of 3000 users we had the following results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	3000	704	1215	8743	0.033%	21.30107
home	45000	957	1399	15332	0.036%	147.58889
codebook	15000	1083	1552	15343	0.020%	52.29506
open_assignment	30000	1005	1477	15310	0.047%	101.79015
save_code	60000	957	1407	15307	0.048%	202.80273
TOTAL	153000	974	1422	15320	0.041%	498.49799

TABLE 3.19: 3000 users on 16 Node workers

### 3.4.5 Comparative Results

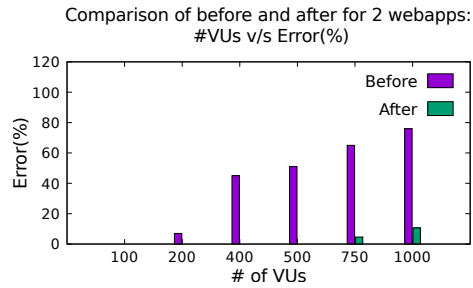


FIGURE 3.30: Comparison of before and after for 2 webapps

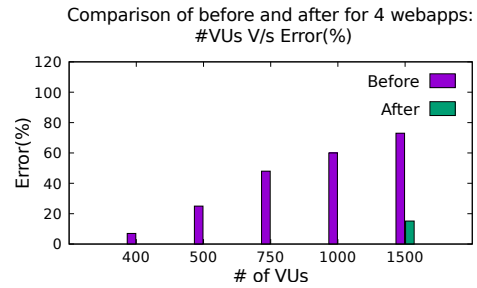


FIGURE 3.31: Comparison of before and after for 4 webapps

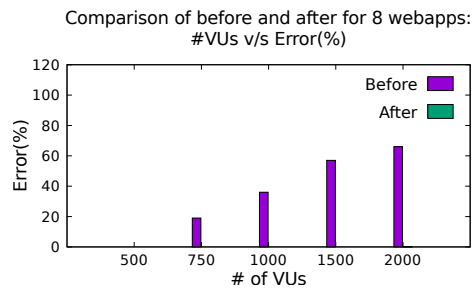


FIGURE 3.32: Comparison of before and after for 8 webapps

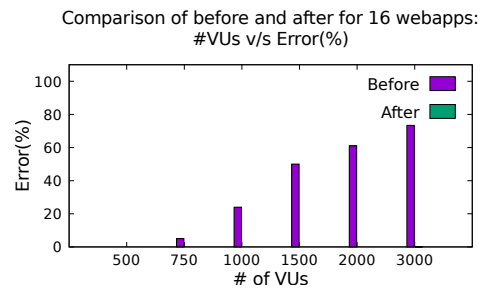


FIGURE 3.33: Comparison of before and after for 16 webapps

Here each graph shows a comparison of WebApp performance for the same number of Node workers. The comparison is between the baseline results and results after our changes on increasing user load. The violet line indicates baseline, whereas the green line indicates our results. In each case we can see huge difference between these 2 lines. After our changes the error% almost went down to 0.

## Chapter 4

# Database Analysis

As Prutor is a database intensive application, a seperate chapter has been dedicated to the database analysis. All the user data is stored in MySQL and all the non-user data i.e. environment settings for compilation and other metadata has been stored in MongoDB. Both of these databases have their seperate containers by the name of **rdb** and **nosql** respectively. Since, the user data forms the major component, we have analysed that in this chapter. There are 2 sections in this chapter. In the first section we have described Query level analysis and our modifications. The second section has been dedicated to Parameter Tuning.

### 4.1 Query Analysis

In this section, we have gone a level down from the above talked features and their respective HTTP requests (Sec. [3.1.1](#)). Here we talk about their APIs and SQL queries. All the analysis has been done with the help of **MySQL-workbench** (Sec. [2.2](#)). For detailed step-by-step instructions for setting up, refer Chap. [B](#).

We have analyzed each query from different perspectives. These perspectives are Index Usage, Queries doing sorting, Queries doing full table scans and Queries using Temporary tables. Each of these perspective has been dedicated a subsection.

### 4.1.1 Index Usage

For checking index usage on a MySQL server, the most common tool is **explain** [35]. MySQL-Workbench provides explain tool in both tabular and visual format. We have used tabular format. The noteworthy fields in the explain tabular output are have been briefly described below.

- possible\_keys - These are the columns which MySQL plans to use for row filtering.
- key - These are the columns which MySQL actually decided to use for filtering.
- rows - It is an estimate of the number of rows that might be extracted
- filtered - It is estimated percentage of the rows that will be filtered out of the total number of rows of the table.
- Extra - It contains information about the features which MySQL used to resolve the query. Exhaustive list of values that can come in this column can be found at [35].

#### • GET /home

##### – events.getOngoingEvents

This API fetches the ongoing events of a user. It hits the mysql server with the following query.

```
SELECT event.id AS event_id , TYPE , NAME ,time_start, time_stop FROM
event INNER JOIN schedule ON event.id=schedule.event_id WHERE event.is_deleted
= 0 AND schedule.is_deleted = 0 AND '2018-03-06 22:53:00' BETWEEN time_start
AND time_stop AND schedule.id IN
( SELECT schedule_id FROM slot WHERE slot.is_deleted=0 AND section =
( SELECT section FROM account WHERE account.is_deleted = 0 AND 'id' =
'd30f7f9d-bf62-4934-b09a-0d9b10651460' ));
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	PRIMARY	schedule	ALL	PRIMARY,event_id			3	33.33	Using where
1	PRIMARY	event	eq_ref	PRIMARY	PRIMARY	its.schedule.event_id	1	10.00	Using where
1	PRIMARY	slot	ref	schedule_id	schedule_id	its.schedule.id	4	8.33	Using where; FirstMatch(event)
3	SUBQUERY	account	const	PRIMARY	PRIMARY	const	1	100.00	

FIGURE 4.1: Explain:getOngoingEvents

As we can see in Fig. 4.1 that only schedule table does not use index. It is because the data with which performance evaluation has been done is the dummy

data and only contains 10 rows in the schedule column, so query optimizer might have chosen to ignore the index and do a row by row scan.

The next query that this API uses is to fetch the questions of the event fetched from the above query.

```
SELECT id,question,is_submitted,max_marks FROM assignment WHERE as-
signment.is_deleted=0 AND user_id='d30f7f9d-bf62-4934-b09a-0d9b10651460' AND
event_id=2 ORDER BY question
```

id	select_type	table	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	assignment	event_id,user_id	user_id	const	30	5.00	Using index condition; Using where; Using filesort

FIGURE 4.2: Explain:Get Questions in the Ongoing Event

Figure 4.2 we can see that the query uses sorting.

#### – statistics.getCourseStatistics

This API calculates the course statistics such as assignments submitted, assignments remaining and number of events attended. It executes the following query.

```
SELECT event_id, assignment.id, is_submitted, TYPE FROM its.assignment
INNER JOIN its.event ON assignment.event_id=event.id WHERE event.is_deleted
= 0 AND assignment.is_deleted = 0 AND event_id IN
( SELECT event_id FROM its.schedule WHERE schedule.is_deleted = 0 AND
id IN
( SELECT schedule_id FROM its.slot WHERE slot.is_deleted = 0 AND section
=
( SELECT section FROM its.account WHERE account.is_deleted = 0 AND id
= 'd30f7f9d-bf62-4934-b09a-0d9b10651460' )
)
AND time_start < '2018-06-06 22:53:00' )
AND user_id ='d30f7f9d-bf62-4934-b09a-0d9b10651460';
```

id	select_type	table	type	key	ref	rows	filtered	Extra
1	PRIMARY	schedule	ALL			3	33.33	Using where; Start temporary
1	PRIMARY	event	eq_ref	PRIMARY	its.schedule.event_id	1	10.00	Using where
1	PRIMARY	slot	ref	schedule_id	its.schedule.id	4	8.33	Using where
1	PRIMARY	assignment	ref	user_id	const	30	5.00	Using where; End temporary
4	SUBQUERY	account	const	PRIMARY	const	1	100.00	

FIGURE 4.3: Explain:getCourseStatistics

Fig. 4.3 shows that all the tables involved are using indexes. One interesting thing to note here is the values **Start Temporary** and **End Temporary**. This indicates the usage of temporary tables which should be dealt with.

– **grading.getScoreCard**

This API retrieves the grade history of a user with the help of the following query.

```
SELECT id ,event_id,event_name , question , score , max_marks , is_submitted
FROM assignment WHERE is_deleted = 0 AND user_id = 'd30f7f9d-bf62-4934-
b09a-0d9b10651460' AND event_id IS NOT NULL
```

id	select_type	table	type	key	ref	rows	filtered	Extra
1	SIMPLE	assignment	ref	user_id	const	30	5.00	Using where

FIGURE 4.4: Explain:getScoreCard

Fig 4.4 shows the usage of user\_id column as index.

• **GET /codebook**

– **assignments.getCodebook**

Codebook is an arena in Prutor, where a user can see his/her previously submitted codes for practice as well as assignment questions. This API enables a user to view his/her codebook with the help of the following query.

```
SELECT env , assignment.id , problem_id , title , category , is_practice ,
event_id , event_name , question , is_submitted FROM assignment INNER JOIN
problem ON assignment . problem_id = problem . id WHERE assignment .
is_deleted = 0 AND problem . is_deleted = 0 AND user_id = 'd30f7f9d-bf62-
4934-b09a-0d9b10651460' AND event_id IN
( SELECT id FROM event WHERE event . is_deleted = 0 AND NOT EXISTS
( SELECT id FROM schedule WHERE schedule . is_deleted = 0 AND sched-
ule . event_id = event . id GROUP BY id HAVING '2018-03-06 22:53:00'
<max(time_stop)
)
) OR event_id IS NULL;
```

id	select_type	table	type	key	rows	filtered	Extra
1	PRIMARY	assignment	ref	user_id	30	10.00	Using where
1	PRIMARY	problem	eq_ref	PRIMARY	1	10.00	Using where
2	DEPENDENT SUBQUERY	event	unique_subquery	PRIMARY	1	10.00	Using where
3	DEPENDENT SUBQUERY	schedule	ref	event_id	1	33.33	Using index condition; Using where; Using temporary; Using filesort

FIGURE 4.5: Explain:getCodebook

As we can see in Figure 4.5, all the tables are using indexes. Apart from this, the Extra field is also worth noticing. It says that this query uses sorting and temporary tables.

- **GET /editor/id**

- **assignments.getLastSavedCode**

Prutor saves multiple versions of a code for an assignment. Each version can be retrieved on the basis of a code id or timestamp. This API fetches the latest saved code from the history of codes for that assignment. The following is the query that it uses.

```
SELECT id , user_id , contents , save_time FROM CODE WHERE code .  
is_deleted = 0 AND assignment_id = 123563 ORDER BY save_time DESC  
LIMIT 1
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	code	ref	assignment_id	assignment_id	5	const	1	10.00	Using index condition; Using where; Using filesort

FIGURE 4.6: Explain:getLastSavedCode

Figure 4.6 shows the usage of sorting by the query.

- **assignments.getAssignmentProblem**

As the name suggests, this API fetches the problem associated with a particular assignment. Following is the query which does this.

```
SELECT statement,env FROM problem WHERE problem.is_deleted=0 AND  
id=(SELECT problem_id FROM assignment WHERE assignment.is_deleted=0  
AND id=123563)
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	PRIMARY	problem	const	PRIMARY	PRIMARY	const	1	100.00	
2	SUBQUERY	assignment	const	PRIMARY	PRIMARY	const	1	100.00	

FIGURE 4.7: Explain:getAssignmentProblem

- **assignments.getAssignmentDetails**

This API fetches the details like question number, maximum marks, event to which that particular assignment belongs. Following is the query which does this job.

```
SELECT event_name,question,max_marks FROM assignment WHERE is_deleted=0  
AND id=123563
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	assignment	const	PRIMARY	PRIMARY	const	1	100.00	

FIGURE 4.8: Explain:getAssignmentDetails

#### – events.isAllowed

This API checks that whether the problem that is being shown in this event, is not a practice problem. If it is a practice problem, then a student cannot access it as an assignment. If it is not a practice problem, then it checks the current time with the stop time of the event. If the current time exceeds the stop time, then again the user cannot open that assignment.

*SELECT is\_practice FROM problem WHERE problem . is\_deleted = ? AND id = ( SELECT problem\_id FROM assignment WHERE assignment . is\_deleted = ? AND id = ? )*

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	PRIMARY	problem	const	PRIMARY	PRIMARY	const	1	100.00	
2	SUBQUERY	assignment	const	PRIMARY	PRIMARY	const	1	100.00	

FIGURE 4.9: Explain:isAllowed

Figure 4.7, 4.8 and 4.9 shows the usage of primary key as indexes.

### 4.1.2 Queries doing Sorting

With Sorting						
List all normalized statements that have done sorts, access in the following priority order - sort_merge_passes, sort_scans and sort_rows						
Query	Executed (#)	Sorts Using Scans (#)	Sorts Using Range (#)	Rows Sorted (#)	▲ Avg Rows Sorted (#)	
SELECT `id`, `question`, `is ... t_id` = ? ORDER BY `question`	14961	0	14951	149500	10.000000	
SELECT `env`, `assignment` ... )) OR `event_id` IS NULL)	4979	149040	0	149040	30.000000	
SELECT `id`, `user_id`, `con ... R BY `save_time` DESC LIMIT ?	9993	0	9990	9990	1.000000	
SELECT IF ((`locate` (? , ... . `COMPRESSED_SIZE`)) DESC	1	4	0	4170	4170.000000	
SELECT `performance_schema` . ... _index_usage` . `OBJECT_	8	8	0	223	28.000000	
SELECT COUNT (?) AS `cnt`, `r ... _by_digest` GROUP BY `avg_t	1	4	0	68	68.000000	
SELECT `performance_schema` . ... IMER_WAIT` AS `avg_laten	1	1	0	31	31.000000	
SELECT `sys` . `format_stateme ... l` (( `performance_schema`	1	1	0	8	8.000000	
SELECT `performance_schema` . ... ex_usage` . `COUNT_READ	1	1	0	1	1.000000	

FIGURE 4.10: Queries doing Sorting

Only the top 3 queries highlighted in Fig. 4.10 are relevant to our schema. These 3 queries are mentioned above in Sec. 4.1.1 in the APIs **events.getOngoingEvents** in Sec. 4.1.1(2<sup>nd</sup> query), **assignments.getCodebook** in Sec. 4.1.1 and the API **assignments.getLastSavedCode** in Sec. 4.1.1. Here references to sections might be showing the same number, but when clicked, they will lead to different sections.

The fix for the usage of sorting by 2 APIs `events.getOngoingEvents` and `assignments.getCodebook` has been discussed in Sec. 4.2. In order to fix for API `assignments.getLastSavedCode` we changed the indexes of the code table. Earlier there were 2 single indexes.

- `user_id`
- `assignment_id`

Now, here filtering is happening on the column `assignment_id`, but sorting is happening on column `save_time`. So if we include `save_time` in our index, we would not need to explicitly sort the rows, as they would already be sorted in the order of the indexed attribute. So now the new indexes are as follows.

- `user_id`
- `assign_savedtime(assignment_id,save_time)`

### 4.1.3 Queries doing full table scans

Full Table Scans			
Lists statements that have performed a full table scan. Access query performance and the where clause(s) and if no index is used, consider adding indexes for large tables			
Query	Executed (#)	No Index Used	No Good Index Used
SELECT 'event_id', 'assignment_id', 'rt' < NOW ) AND 'user_id' = ?	14998	14998	14988
SELECT 'event_id', 'id' AS 'even_id', 'deleted' = ? AND 'id' = ? )	14973	14973	14961
SELECT 'performance_schema' . ... '_index_usage' . 'OBJECT_NAME'	8	8	8
SELECT 'st' . * FROM 'performance_schema' . 'st' . 'nesting_event_id' = ?	6	6	6
SELECT 'st' . * FROM 'performance_schema' . 'st' . 'nesting_event_id' = ?	6	6	6

FIGURE 4.11: Queries doing full table scan

The queries that are shown in Fig. 4.11 are described in Sec. 4.1.1 and 4.1.1. The fix for these queries has been discussed in Sec. 4.2.

### 4.1.4 Queries Using Temporary Tables

Using Temp Tables						
Lists all statements that use temporary tables - access the highest # of disk temporary tables, then memory temp tables						
Query	Executed (#)	Temp Tables	Temp Tables	Avg. Temp T	Percent Temp	Digest
SELECT 'event_id', 'assignment_id', 'rt' < NOW ) AND 'user_id' = ?	14998	14987	0	1.000000	0.000000	0caf0ff6329938e1a01e7d615eaa418f
SHOW GLOBAL STATUS	10818	10818	0	1.000000	0.000000	011da6fa0d4fdfe1518ad83484e620e1
SELECT 'env', 'assignment_id', 'rt' < NOW ) OR 'event_id' IS NULL )	4979	4960	0	1.000000	0.000000	576499b14faf26c83de828aa9a31705a
SHOW SESSION VARIABLES LIKE ?	16	16	0	1.000000	0.000000	932d3da3dd875d82acd93e67e96996a

FIGURE 4.12: Queries using temporary tables

The queries that are shown in Fig. 4.12 have been described in Sec. 4.1.1 and 4.1.1. The fix for these have been discussed in Sec. 4.2.

## 4.2 Usage of Materialized Views

We have seen queries in section 4.1.1, 4.1.1 and 4.1.1 use temporary tables, do on-the-fly sorting or do a full table scan. These queries are written in such a way that requires lot of processing at the time of their execution. So we needed a solution through which we can pre-process our required data(pertaining to these queries) before their execution. The answer that came to our mind was Materialized Views[36].

In the version that we are using i.e. MySQL 5.7, there is no concept of Materialized Views. However, one can imitate the behaviour of Materialized Views with the help of **Stored Procedures**[37] and **Triggers**[38].

### 4.2.1 ongoingAssignments\_mv

We have made use of 2 MVs. First, was to materialise the data for queries in API **events.getOngoingEvents** and **statistics.courseStatistics**. Since most of the attributes in both of the queries were common, so we took a union of these attributes and formed a single MV.

```

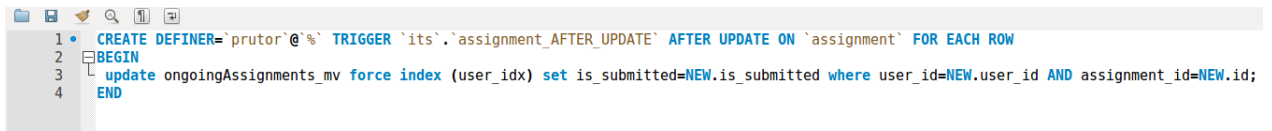
1 • CREATE DEFINER='prutor'@'%' PROCEDURE `materialize_ongoingAssignments`()
2 BEGIN
3 DROP TABLE IF EXISTS ongoingAssignments_mv;
4 CREATE TABLE ongoingAssignments_mv(
5     id int(11) NOT NULL auto_increment,
6     user_id varchar(50) DEFAULT NULL,
7     event_id int(11) DEFAULT NULL,
8     name varchar(255) DEFAULT NULL,
9     assignment_id int(11) DEFAULT NULL,
10    question int(11) DEFAULT NULL,
11    is_submitted tinyint(1) NOT NULL DEFAULT '0',
12    max_marks int(11) DEFAULT NULL,
13    type varchar(255) DEFAULT NULL,
14    time_start timestamp NULL DEFAULT CURRENT_TIMESTAMP,
15    time_stop timestamp NULL DEFAULT CURRENT_TIMESTAMP,
16    PRIMARY KEY (id)
17 ) ENGINE=InnoDB;
18
19 INSERT INTO ongoingAssignments_mv(user_id,event_id,name,assignment_id,question,is_submitted,max_marks,type,time_start,time_stop)
20 SELECT user_id,assignment.event_id,name,assignment.id,question,is_submitted,max_marks,type,time_start,time_stop
21 FROM assignment INNER JOIN event
22 ON assignment.event_id=event.id
23 INNER JOIN schedule
24 ON event.id = schedule.event_id
25 WHERE event.is_deleted = 0 AND assignment.is_deleted=0 AND schedule.is_deleted = 0 AND CURRENT_TIMESTAMP BETWEEN time_start AND time_stop
26 ORDER BY question;
27 END

```

FIGURE 4.13: MV:ongoingAssignments

Every time a student submits a code, his/her number of submissions should be reflected in the home page view. This all depends on the **is\_submitted** attribute of the assignment

table. But we are fetching from our view. So we needed a mechanism to update the rows of our view as and when the `is_submitted` attribute is getting updated. That is when **triggers** came into picture.



```

1 • CREATE DEFINER='prutor'@'%' TRIGGER `its`.`assignment_AFTER_UPDATE` AFTER UPDATE ON `assignment` FOR EACH ROW
2 BEGIN
3   update ongoingAssignments_mv force index (user_idx) set is_submitted=NEW.is_submitted where user_id=NEW.user_id AND assignment_id=NEW.id;
4 END

```

FIGURE 4.14: MV:Trigger for updating ongoingAssignment MV

In the figure 4.14, we have written an **After update** trigger on the **assignment** table. This is because, when code is submitted, the row corresponding to the `assignment_id` is updated with the `is_submitted` value of 1. So now after each update in the assignment table, our MV shown in figure 4.13 will be updated.

## 4.2.2 codebook\_mv

This MV has been used to fix the issues with query in the API `assignments.getCodebook` in section 4.1.1.



```

1 • CREATE DEFINER='prutor'@'%' PROCEDURE `materialize_codebook`()
2 BEGIN
3   DROP TABLE if exists codebook_mv;
4   CREATE TABLE codebook_mv (
5     id int(11) NOT NULL auto_increment,
6     user_id varchar(50) DEFAULT NULL,
7     env varchar(100) NOT NULL DEFAULT 'C',
8     assignment_id int(11) DEFAULT NULL,
9     problem_id int(11) DEFAULT NULL,
10    title varchar(255) DEFAULT NULL,
11    category varchar(255) DEFAULT NULL,
12    is_practice tinyint(1) NOT NULL DEFAULT '0',
13    event_id int(11) DEFAULT NULL,
14    event_name varchar(255) DEFAULT NULL,
15    question int(11) DEFAULT NULL,
16    is_submitted tinyint(1) NOT NULL DEFAULT '0',
17    PRIMARY KEY (id),
18    KEY user_idx(user_id)
19  ) ENGINE=InnoDB;
20
21  INSERT INTO codebook_mv(user_id,env,assignment_id,problem_id,title,category,is_practice,event_id,event_name,question,is_submitted)
22  SELECT user_id,env,assignment_id,problem_id,title,category,is_practice,event_id,event_name,question,is_submitted
23  FROM assignment INNER JOIN problem ON assignment.problem_id = problem.id WHERE
24  (
25    event_id IN
26    (
27      SELECT id FROM event WHERE NOT EXISTS
28      (
29        SELECT event_id FROM schedule WHERE schedule.event_id = event.id AND schedule.is_deleted=0
30        GROUP BY event_id HAVING CURRENT_TIMESTAMP < max(time_stop)
31      )
32    )
33    OR event_id IS NULL
34  );
35  END

```

FIGURE 4.15: MV:codebook

Here the attributes in this MV are based on the attributes in the query in section 4.1.1.

### 4.3 Comparative Results

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	1000	16521	29570	32539	0.000%	12.61209
home	15000	2461	1685	28386	0.000%	100.16761
codebook	5000	2236	1721	26893	0.000%	36.40520
open_assignment	10000	1588	1716	2045	0.000%	75.57208
save_code	20000	1344	1669	2015	0.000%	187.65071
TOTAL	51000	2106	1719	25842	0.000%	339.94561

TABLE 4.1: 1000 users results before DB changes

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	1000	16183	29181	31960	0.000%	12.70793
home	15000	2424	1634	27859	0.000%	100.62117
codebook	5000	2184	1725	26160	0.000%	37.20515
open_assignment	10000	1561	1627	2230	0.000%	70.65041
save_code	20000	1383	1630	2264	0.000%	144.47318
TOTAL	51000	2093	1660	25523	0.000%	339.45687

TABLE 4.2: 1000 users results after DB changes

### 4.4 Parameter Tuning

In this section, we have described the parameters of mysql server that we found were necessary to be tuned for better performance. InnoDB buffer pool was the main feature around which our parameter tuning revolved.

**InnoDB buffer pool** is the memory space that holds many in-memory data structures of InnoDB, buffers, caches, indexes and even row-data. MySQL offers a wide range of parameters, that can be tuned for high-performance. Some of them are related to innodb, some of them are general. The ones we touched upon are listed below.

#### 4.4.1 innodb\_buffer\_pool\_size

The default value of `innodb_buffer_pool_size` is 128M. As per [39] and [40] if we have a dedicated server for database, we can give it 70-80% of our RAM, but in Prutor, every container runs on the same machine. So we need to follow a more calculated approach.

- **Calculating the size of the database**

The following query suggested by [40] reads the size of data and indexes of the `innodb` tables from the `information_schema` and takes in account a 60% overhead in maintaining in-memory buffers.

```
SELECT CEILING
(Total_InnoDB_Bytes*1.6/POWER(1024,3)) RIBPS FROM
(SELECT SUM(data_length+index_length) Total_InnoDB_Bytes
FROM information_schema.tables WHERE engine='InnoDB' ) A;
```

For our dummy data, it gave us **1GB**.

- **Calculating the memory requirement overall**

Now if we imagine extreme scenario for our machine (8cores,16GB RAM), we will have **8 node workers** and **~300 apache workers** (discussed in chapter 4). So in order to see the memory consumption, we executed these many processes and executed `ps_mem.py`, which gave us **5.5GB**(including `mysqld`).

So this leaves us with 10.5GB of RAM. Now ideally for our data, pool size of 2GB should be enough, but we will be using `buffer_pool_instances` and the recommended setting suggested by [41] is that each buffer pool should get atleast 1GB and for greater concurrency `buffer_pool_instances` should be atleast 4. So we found **6GB** as an appropriate value in our case.

#### 4.4.2 innodb\_buffer\_pool\_instances

This parameter divides the buffer pool into small regions. For this to be enabled, the buffer pool size should be atleast 1GB. Its usage replaces a single large buffer pool with multiple small buffer pools, which improves the concurrency. Another factor which contributes to increase in concurrency is that each buffer pool instance has its own buffer pool mutex. We have set it to **4**.

### 4.4.3 Escape from Swap

A Server running database should never use swap memory. We found that due to less memory allocated to buffer pool, mysqld was using swap memory. This was rectified by doing 2 things. First was increasing buffer pool. Second was tuning a kernel parameter **vm.swappiness**. It's value ranges from 0 to 100, 0 being the least aggressive and 100 being the most aggressive. By default it is set to 60. Sites like [\[34\]](#) suggested **30** to be an appropriate value.

### 4.4.4 Query Cache

Query cache stores the select statements along with their results. If the server receives the same query later, rather than executing the query, it serves the result from the query cache. There are 2 essential parameters that cover its tuning.

- **query\_cache\_type**- should be set to ON.
- **query\_cache\_size**- [\[41\]](#) recommends it to be in tens of MBs, so we went with 10M.

### 4.4.5 skip\_name\_resolve

To skip the DNS lookup. By default it is set to OFF, i.e. DNS lookup is enabled. In order to disable it, it should be set to ON.

## Chapter 5

# Engine: Analysis and Benchmarking

Engine component is the back-end server, responsible for handling compilation, execution and evaluation HTTP requests. It runs on **apache** server and is written in **PHP**.

### 5.1 Event Workload

The compilation, execution and evaluation workload can be sent from all 3 perspectives (Sec. 3.1), but we will only consider the ones sent from event perspective, just like in Chap. 3. The tables below show the anatomy of all 3 requests.

<b>Use Case</b>	Compilation
<b>HTTP Request URL</b>	/compile
<b>Request Type</b>	POST
<b>Parameters</b>	- assignment_id - code - env

TABLE 5.1: HTTP Post Request for Compilation

<b>Use Case</b>	Execution
<b>HTTP Request URL</b>	/execute
<b>Request Type</b>	POST
<b>Parameters</b>	<ul style="list-style-type: none"> <li>- assignment_id</li> <li>- code</li> <li>- env</li> <li>- testcase</li> </ul>

TABLE 5.2: HTTP Post Request for Execution

<b>Use Case</b>	Evaluation
<b>HTTP Request URL</b>	/evaluate
<b>Request Type</b>	POST
<b>Parameters</b>	<ul style="list-style-type: none"> <li>- admin</li> <li>- assignment_id</li> </ul>

TABLE 5.3: HTTP Post Request for Evaluation

## 5.2 Baselining Scalability of Engine

### 5.2.1 Setup

Refer Sec. [3.2.1](#)

### 5.2.2 Workload Specification

Here we have Login, compilation, execution and evaluation HTTP Requests as part of our test plan. The anatomy of these requests has been covered in Sec. [5.1](#). Rest all is same as mentioned in Sec. [3.2.2](#).

### 5.2.3 Performance Metrics

Like WebApp, the performance metrics is **Error%** here, with a difference that in WebApp we were taking the Error% of Login Request only, here we have considered the average Error% i.e. average of all the HTTP Request elements in the JMeter Test Plan. The

reason behind this is that in the WebApp, if a particular VU was not able to login, its other HTTP requests were getting redirected. The redirect response is not counted as error, but in the case of engine, if a VU is not able to login, it will not be allowed access to compilation and other APIs, thereby resulting in 403 Forbidden HTTP Response.

#### 5.2.4 Results

Results have been shown in comparison with the results after modifications in Sec. [5.5.2](#).

### 5.3 Parameter Tuning

This section describes how we identified the correct parameters, that led us hit a scale of **1000** users with single engine container. Like Chap. [3](#), here also **server maxconn** played a key role. Also, another proxy parameter **contimeout** came into the picture. Now, as we were scaling an apache server, the key parameters of Apache server like **MaxRequestWorkers** also came into play. These parameters have been briefly described below.

- **MaxRequestWorkers** - It is the number of Apache workers that will run simultaneously inside Engine. This parameter directly controls the number of client connections that will be served simultaneously.
- **MaxConnectionsPerChild** - It is the maximum number of connections that an Apache worker can handle. It's default value is 1000. This means that after handling 1000 connections, the child will terminate. To prevent this, it is preferred that it should be set 0. This removes any upper limit on this parameter.
- **ServerLimit** - It sets the upper bound on the configurable value of MaxRequestWorkers.
- **contimeout** - It defines the maximum time, the proxy will wait for a connection attempt to back-end server.
- **server maxconn** - explained in Sec. [3.4](#)

### 5.3.1 Reaching 500 users

- **100 users**

Without any tuning, on executing the test plan for 100 users, we got an error% of **12.75%** Increasing **server maxconn** to **100** solved the problem.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	100	189	234	247	0.000%	2.10402
compile	500	4072	6586	8041	0.000%	3.73703
execute	500	3786	5868	7660	0.000%	3.68878
evaluate	500	5113	7589	9276	0.000%	3.63446
TOTAL	1600	4065	6829	8629	0.000%	11.08955

TABLE 5.4: 100 users on 10 Apache workers

- **200 users**

At this time, we had server maxconn at 100 and MaxRequestWorkers at 10. On this configuration, we got the error rate of **4%**. We then increased the MaxRequestWorkers to **20**, which solved the problem.

- **400 users**

We now had **server maxconn** at 200 and **MaxRequestWorkers** at 20. On Executing the test plan for 400 users on these conditions we got **52%** error%. So we raised **MaxRequestWorkers** to **50** and the error% got reduced to **3%**.

In WebApp we saw that the appropriate value of server maxconn was approximately equal to the number of users in the workload. So we tried that ideology here and got even worse results. The error% went up to 40%. Here are the results.

Label	# Samples	Average	90% Line	99% Line	Max	Error %	Throughput
HTTP Request	400	4623	8447	9534	9732	0.000%	6.96512
compile	2000	4066	9198	30172	56024	38.200%	13.67549
execute	2000	3464	8145	30178	53033	43.050%	13.68420
evaluate	2000	4981	11810	33651	56014	49.100%	13.63447
TOTAL	6400	4198	9516	30181	56024	40.734%	41.67643

TABLE 5.5: 400 users on Engine with server maxconn 400

Compared to WebApp, Engine is handling more write-intensive queries. After accepting the connection request from proxy, it compiles the code, writes it into the

database and then sends the processed feedback as response. So, when we raise server maxconn to **400**, the back-end server is flocked by the HTTP Requests, more than it can handle and hence a greater chunk of requests fail.

Due to the nature of the requests being handled by the Engine, we considered increasing the contimeout value to **50000**. To test it's contribution, we re-ran the same test and got error% reduced to 20%. Here are the results.

Label	# Samples	Average	90% Line	99% Line	Max	Error %	Throughput
HTTP Request	400	4522	8394	9156	9234	0.000%	7.02284
compile	2000	5426	10426	21831	53046	19.800%	11.11519
execute	2000	5635	10914	31319	58025	21.700%	11.12712
evaluate	2000	7336	13474	42387	58025	27.300%	11.14703
TOTAL	6400	6032	11664	31959	58025	21.500%	34.32666

TABLE 5.6: engine:400 users on Engine with contimeout 50000

So even though server maxconn is set higher than it should be, the tuning of contimeout parameter yields better results. Hence we kept it tuned for future testing.

Finally, we reset the server maxconn to 100 and got the following results at 50 apache workers.

Label	# Samples	Average	90% Line	99% Line	Max	Error %	Throughput
HTTP Request	400	5187	9718	10547	10692	0.000%	6.78196
compile	2000	6476	10408	13460	14537	0.000%	10.86195
execute	2000	6210	9003	12991	14216	0.000%	10.70761
evaluate	2000	8392	11718	16583	17059	0.000%	10.64192
TOTAL	6400	6911	10644	15207	17059	0.000%	32.84713

TABLE 5.7: engine:400 users on Engine with server maxconn 100

Now, if you notice the Maximum Response Time has drastically gone down. This is because the response time calculation starts after a **session** [42] has been established. A session comprises of 2 connections, one from the client to HAProxy, and the other from HAProxy to the appropriate backend server. So in the case of 400 server maxconn, session was formed, but the client had to wait as the server was not able to finish up with the requests. In the case of 100 server maxconn, the proxy is restricting the client from forming a session, which gives our back-end server enough time to finish up the requests that it already has.

- **500 users** So till now we had server maxconn at 100 and MaxRequestWorkers at 50. On running the test plan for 500 users on these parameter settings, we got no errors, but ART was too damn high to be of practical use.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	500	6757	12327	13536	0.000%	8.03303
compile	2500	9057	13468	15853	0.000%	11.50822
execute	2500	8270	11900	15406	0.000%	11.22254
evaluate	2500	10982	15456	17723	0.000%	11.00468
TOTAL	8000	9269	13625	17201	0.000%	33.98225

TABLE 5.8: engine:500 users on 50 MaxRequestWorkers

So we raised MaxRequestWorkers to 100 and got the following results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	500	7277	13658	15095	0.000%	7.86819
compile	2500	6169	8754	10855	0.000%	13.44701
execute	2500	5669	7976	10060	0.000%	13.41662
evaluate	2500	8696	12092	14138	0.000%	13.33860
TOTAL	8000	6872	10440	13727	0.000%	40.90900

TABLE 5.9: 500 users on 100 MaxRequestWorkers

### 5.3.2 Reaching 1000 users

- **750 users** For 750 users, we tried with 200 MaxRequestWorkers and server maxconn 100. Though the error% was 0 but here also ART was too high, 9secs for compilation request and 12 secs for evaluation request. So we tried with server maxconn 200 and MaxRequestWorkers 200, which gave the following results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	750	14027	26275	28962	0.000%	9.62168
compile	3750	8030	10320	11913	0.000%	16.21383
execute	3750	7791	9978	11565	0.000%	16.06340
evaluate	3750	12223	16497	19060	0.000%	15.99529
TOTAL	12000	9640	15259	24478	0.000%	49.48637

TABLE 5.10: 750 users on 200 MaxRequestWorkers

We had to settle with this value of ART. This is the lowest ART that we were able to achieve for 750 users.

- **1000 users** With 250 MaxRequestWorkers and 200 server maxconn we got the following results.

Label	# Samples	Average	90% Line	99% Line	Error %	Throughput
HTTP Request	1000	21544	39184	43110	0.000%	10.86213
compile	5000	11493	14671	16051	0.000%	17.02319
execute	5000	11035	14020	15285	0.000%	16.88419
evaluate	5000	15173	20231	23232	0.000%	16.83717
TOTAL	16000	13128	19154	36584	0.000%	52.52584

TABLE 5.11: 1000 users on 250 MaxRequestWorkers

As we can see there is no error% but the ART of each request is above 10 seconds. So we tried with 300 MaxRequestWorkers, but got the same result.

## 5.4 Other modifications

### 5.4.1 ORM Removal

Prutor uses **RedBean** [43] in the engine component for Database access. RedBean is an ORM [44]. So first we should know what is ORM and why it is used.

The purpose of **Object Relational Mapper** (ORM) is to provide an intermediate layer between 2 incompatible type-systems in an object-oriented environment. In the context of database intensive application, these incompatible type-systems are data store and programming objects. So, concisely we can say that ORMs maps a relational database into an object which can be understood by the application's programming environment. Some of the most common features that ORM provides are.

- **Handling Database Access** ORM provides APIs for everything from connecting to executing CRUD queries. So a developer does not need to write SQL code himself/herself.
- **Abstraction of the Database** This point is a consequence of the previous point. It is because of ORM APIs that developer does not need to bother about which database has been used. ORM handles everything.

- **Security** Provides security against SQL injection.

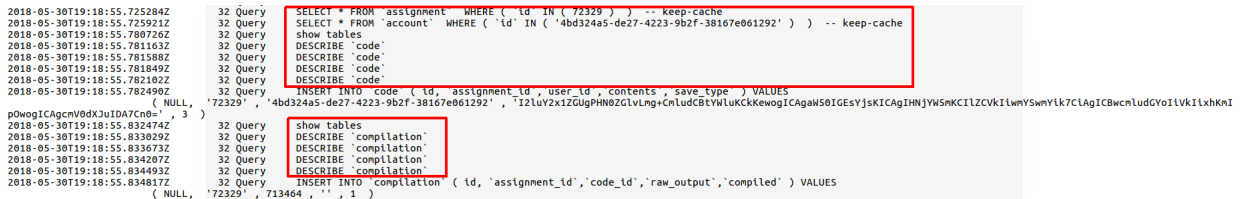
These features might sound fancy, but are of very little use in Prutor, especially in the engine component. It is because of the following reasons.

- Engine mostly handles insertion queries. For a developer, writing SQL insert queries, should not be a hairy task. There is not much to optimize here. Hence using an ORM for database access in our case is clearly an overkill.
- There are many lightweight database connectors which provide security against SQL injection. For instance, php's native mysql connectors also provides a method in which we can send prepared [45] statements to a MySQL server.

In general also, ORM's advantages are it's biggest disadvantages. Due to the abstraction of task of query writing, we cannot optimize our queries. In addition to this the biggest disadvantage of ORM is execution of extra queries, which has been discussed in the next subsection.

#### 5.4.1.1 Extra Queries

The main drawback of ORM that bugged us was the execution of unnecessary queries before execution of relevant queries. We enabled **general\_log** system variable on our mysql server and gathered the following logs for compilation, execution and evaluation queries.



```

2018-05-30T19:18:55.725284Z 32 Query SELECT * FROM `assignment` WHERE ( `id` IN ( '72329' ) ) -- keep-cache
2018-05-30T19:18:55.725921Z 32 Query SELECT * FROM `account` WHERE ( `id` IN ( '4bd324a5-de27-4223-9b2f-38167e061292' ) ) -- keep-cache
2018-05-30T19:18:55.780726Z 32 Query show tables
2018-05-30T19:18:55.781163Z 32 Query DESCRIBE `code`
2018-05-30T19:18:55.781588Z 32 Query DESCRIBE `code`
2018-05-30T19:18:55.781849Z 32 Query DESCRIBE `code`
2018-05-30T19:18:55.782182Z 32 Query DESCRIBE `code`
2018-05-30T19:18:55.782490Z 32 Query INSERT INTO `code` ( `id`, `assignment_id`, `user_id`, `contents`, `save_type` ) VALUES
( NULL, '72329', '4bd324a5-de27-4223-9b2f-38167e061292', '121uv2x1ZGgPHN0ZGlvdng=CnLudCBTWLUkCkKewogICAgH50IGesYjsKICAgIHJjVW5hNCI1ZCvkIilmYSwyYk7ClAgICBwcnLudGVoIlVklxhKni
pOwogICAgcnVbdXJuIDAtCn0= ', 3 )
2018-05-30T19:18:55.832474Z 32 Query show tables
2018-05-30T19:18:55.833029Z 32 Query DESCRIBE `compilation`
2018-05-30T19:18:55.833673Z 32 Query DESCRIBE `compilation`
2018-05-30T19:18:55.834207Z 32 Query DESCRIBE `compilation`
2018-05-30T19:18:55.834493Z 32 Query DESCRIBE `compilation`
2018-05-30T19:18:55.834817Z 32 Query INSERT INTO `compilation` ( `id`, `assignment_id`, `code_id`, `raw_output`, `compiled` ) VALUES
( NULL, '72329', '713464', '', 1 )

```

FIGURE 5.1: Queries executed during single compilation request

```

2018-05-30T19:21:32.976396Z 49 Query SELECT * FROM assignment WHERE ( 'id' IN ( '72329' ) ) -- keep-cache
2018-05-30T19:21:32.976988Z 49 Query SELECT * FROM account WHERE ( 'id' IN ( '4bd324a5-de27-4223-9b2f-38167e061292' ) ) -- keep-cache
2018-05-30T19:21:33.030699Z 49 Query show tables
2018-05-30T19:21:33.031113Z 49 Query DESCRIBE code
2018-05-30T19:21:33.031540Z 49 Query DESCRIBE code
2018-05-30T19:21:33.031847Z 49 Query DESCRIBE code
2018-05-30T19:21:33.032143Z 49 Query INSERT INTO code ( id, assignment_id, user_id, contents, save_type ) VALUES
2018-05-30T19:21:33.032478Z ( NULL, '72329', '4bd324a5-de27-4223-9b2f-38167e061292', '12LUV2x1ZGugPHN0ZGlvLNg+CnLudCBtYwLUckKewogICAgAHS0IGesYjsKICAgIHhjYw5MKC1LZCVKIwYsWmYk7ClAgICBwcmludGVoIlVklxhkhNl
pwoWIGAgcnV0dXJlIDAtZCne', 3 )
2018-05-30T19:21:33.088711Z 49 Query show tables
2018-05-30T19:21:33.089162Z 49 Query DESCRIBE compilation
2018-05-30T19:21:33.089770Z 49 Query DESCRIBE compilation
2018-05-30T19:21:33.090251Z 49 Query DESCRIBE compilation
2018-05-30T19:21:33.090681Z 49 Query INSERT INTO compilation ( id, assignment_id, code_id, raw_output, compiled ) VALUES
2018-05-30T19:21:33.091180Z ( NULL, '72329', '713463', '1', 1 )
2018-05-30T19:21:33.138821Z 49 Quit
2018-05-30T19:21:33.258097Z 50 Connect prutor@172.17.0.9 on its using TCP/IP
2018-05-30T19:21:33.258242Z 50 Query SET NAMES utf8mb4
2018-05-30T19:21:33.258350Z 50 Query SELECT * FROM assignment WHERE ( 'id' IN ( '72329' ) ) -- keep-cache
2018-05-30T19:21:33.258795Z 50 Query SELECT id,contents FROM code WHERE assignment_id='72329' ORDER BY save_time DESC LIMIT 1
2018-05-30T19:21:33.350248Z 50 Query SELECT id,code_id FROM compilation WHERE assignment_id='72329' ORDER BY compile_time DESC LIMIT 1
2018-05-30T19:21:33.350973Z 50 Query show tables
2018-05-30T19:21:33.351481Z 50 Query DESCRIBE execution
2018-05-30T19:21:33.352066Z 50 Query DESCRIBE execution
2018-05-30T19:21:33.352516Z 50 Query DESCRIBE execution
2018-05-30T19:21:33.352905Z 50 Query DESCRIBE execution
2018-05-30T19:21:33.353370Z 50 Query INSERT INTO execution ( id, code_id, result, input, output ) VALUES
( NULL, '713463', 'OK', '1 2', '2' )

```

FIGURE 5.2: Queries executed during single execution request

```

2018-05-30T19:17:42.340992Z 23 Query SELECT * FROM assignment WHERE ( 'id' IN ( '72329' ) ) -- keep-cache
2018-05-30T19:17:42.341485Z 23 Query SELECT * FROM account WHERE ( 'id' IN ( '4bd324a5-de27-4223-9b2f-38167e061292' ) ) -- keep-cache
2018-05-30T19:17:42.395436Z 23 Query show tables
2018-05-30T19:17:42.395915Z 23 Query DESCRIBE code
2018-05-30T19:17:42.396389Z 23 Query DESCRIBE code
2018-05-30T19:17:42.396766Z 23 Query DESCRIBE code
2018-05-30T19:17:42.397055Z 23 Query DESCRIBE code
2018-05-30T19:17:42.397388Z 23 Query INSERT INTO code ( id, assignment_id, user_id, contents, save_type ) VALUES
2018-05-30T19:17:42.444452Z ( NULL, '72329', '4bd324a5-de27-4223-9b2f-38167e061292', '12LUV2x1ZGugPHN0ZGlvLNg+CnLudCBtYwLUckKewogICAgAHS0IGesYjsKICAgIHhjYw5MKC1LZCVKIwYsWmYk7ClAgICBwcmludGVoIlVklxhkhNl
pwoWIGAgcnV0dXJlIDAtZCne', 3 )
2018-05-30T19:17:42.444251Z 23 Query show tables
2018-05-30T19:17:42.444640Z 23 Query DESCRIBE compilation
2018-05-30T19:17:42.445168Z 23 Query DESCRIBE compilation
2018-05-30T19:17:42.445564Z 23 Query DESCRIBE compilation
2018-05-30T19:17:42.445954Z 23 Query DESCRIBE compilation
2018-05-30T19:17:42.446452Z 23 Query INSERT INTO compilation ( id, assignment_id, code_id, raw_output, compiled ) VALUES
2018-05-30T19:17:42.486380Z ( NULL, '72329', '713463', '1', 1 )
2018-05-30T19:17:42.608189Z 23 Quit
2018-05-30T19:17:42.608192Z 24 Connect prutor@172.17.0.9 on its using TCP/IP
2018-05-30T19:17:42.608405Z 24 Query SET NAMES utf8mb4
2018-05-30T19:17:42.608619Z 24 Query SELECT * FROM assignment WHERE ( 'id' IN ( '72329' ) ) -- keep-cache
2018-05-30T19:17:42.609282Z 24 Query SELECT id,contents FROM code WHERE assignment_id='72329' ORDER BY save_time DESC LIMIT 1
2018-05-30T19:17:42.609265Z 24 Query SELECT env FROM problem WHERE id=(SELECT problem_id FROM assignment WHERE id='72329')
2018-05-30T19:17:42.657665Z 24 Query SELECT id AS test_id,input,output
FROM test_case
WHERE problem_id=(SELECT problem_id FROM assignment WHERE id='72329')
AND is_deleted=0 AND visibility=1
2018-05-30T19:17:42.808847Z 24 Query SELECT id AS test_id,input,output
FROM test_case
WHERE problem_id=(SELECT problem_id FROM assignment WHERE id='72329')
AND is_deleted=0 AND visibility=0 AND type=1
2018-05-30T19:17:42.940856Z 24 Query SELECT env FROM problem WHERE id=(SELECT problem_id FROM assignment WHERE id='72329')
2018-05-30T19:17:42.941280Z 24 Query SELECT id,code_id FROM compilation WHERE assignment_id='72329' ORDER BY compile_time DESC LIMIT 1
2018-05-30T19:17:42.941812Z 24 Query show tables
2018-05-30T19:17:42.967842Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:42.967913Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:42.967913Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:42.968284Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:42.968602Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:42.968971Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:42.969374Z 24 Query INSERT INTO evaluation ( id, assignment_id, compilation_id, code_id, testcase_id, output, result, verdict ) VALUES
2018-05-30T19:17:43.028542Z ( NULL, '72329', '228686', '713463', '1', '2', 'OK', 'ACCEPTED' )
2018-05-30T19:17:43.029079Z 24 Query show tables
2018-05-30T19:17:43.029372Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:43.030293Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:43.030830Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:43.031344Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:43.031912Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:43.032534Z 24 Query DESCRIBE evaluation
2018-05-30T19:17:43.032152Z 24 Query INSERT INTO evaluation ( id, assignment_id, compilation_id, code_id, testcase_id, output, result, verdict ) VALUES
( NULL, '72329', '228686', '713463', '2', '6', 'OK', 'ACCEPTED' )

```

FIGURE 5.3: Queries executed during single evaluation request

The extra queries here have been highlighted in red. The reason behind their execution is object creation of RedBean, which is exactly what ORMs do.

#### 5.4.1.2 Usage of PDO

The PHP Data Objects (PDO) [46] is a lightweight interface for accessing databases in PHP. PDO provides a data-access abstraction layer, rather than database abstraction. This means that there is no mapping of relational tables to objects, no query building, no emulation of missing features. It simply provides a unified API to issue queries and fetch data into objects. Compared to an ORM like RedBean, it has the following advantages.

- **No Extra Queries** As mentioned above that PDO does not involve in object mapping and query building, the queries that are executed on server are only the once that we send.
- **More control over the queries** As PDO only abstracts the data-access layer, it offers more control to developers. A developer can write his/her own queries with their preferred optimization.
- **Better Performance** ORMs are full blown tools for database abstraction. They are built over Database Access Layer (DALs) like PDO. Due to less abstraction involved in PDO, PDOs are much better when it comes to speed and performance.
- **No need to install** PDOs come bundled along with PHP.

In addition to the above benefits, PDOs cover almost all the key functionalities which ORMs provide.

- **Security** PDO has a concept of **Prepared Statements** [45], where a statement is first prepared by putting placeholders for the parameters. PDO supports both positional and named placeholders. These placeholders are then filled with parameter values and the completed query string is sent to server. This process of preparing and executing ensures security against SQL injection.
- **Consistency across databases** This is however not relevant in our case, but the API that PDO provides is consistent across databases. The only that changes is the PDO driver. So as a developer we need not worry about the change in code, if our database is changed.

In the next section, we have discussed about the unnecessary MySQL queries that we removed from the code flow.

### 5.4.2 Removal of unnecessary DB hits

We have called some sql queries as unnecessary because they fall into either of the following categories. We have listed down the queries that we removed under each category.

- Queries that fetched results that could have been provided as POST parameters.

- **Fetching assignment** Before compiling each code of an assignment, the compile API retrieves the respective assignment tuple from the database. It uses the `event_id` field in this tuple and concatenates with the `assignment_id` to name the executable that is built after compilation.

```
// FETCH ASSIGNMENT
$assignment = R::load('assignment', $assignment_id);
$account    = R::load('account',   $assignment->user_id);
$identifier = 'e' . $assignment->event_id . '_' . $assignment_id;
```

FIGURE 5.4: Extra MySQL Query: Fetch assignment

- **Fetching Programming Environment** Evaluation API fetches the programming environment for each request.

We found that the results of these queries could have been easily passed in POST parameters and hence there was no explicit need for their execution.

- Queries that were redundant or single query might have done the job of multiple queries.
- **Fetching Test cases** We found that in the evaluation API there were 2 separate calls for fetching visible and invisible test cases. This could have been easily done by a single query.

```
// Visible test cases.
$query = "SELECT id AS test_id,input,output
FROM test_case
WHERE problem_id=(SELECT problem_id FROM assignment WHERE id=:id)
AND is_deleted=0 AND visibility=1";
$testcases = Helper::cacheData($query, array(':id'=>$assignment_id));
```

FIGURE 5.5: Extra MySQL Query: Fetch visible test cases

```
// Invisible test cases.
$query = "SELECT id AS test_id,input,output
FROM test_case
WHERE problem_id=(SELECT problem_id FROM assignment WHERE id=:id)
AND is_deleted=0 AND visibility=0 AND type=1";
$testcases = Helper::cacheData($query, array(':id'=>$assignment_id));
```

FIGURE 5.6: Extra MySQL Query: Fetch invisible testcases

### 5.4.3 Removal of NoSQL hits

As mentioned in sec 2.1.3, the NoSQL component in Prutor is the container running MongoDB. It stores several collections, only 2 of which are accessed by the engine. One is **env** which stores information like output extension, compile command, source extension and binary extension. The other one is **configs**, which stores information related to the various config parameters such as telemetry, delays and hooks. The data that we just talked about is static data, because it would never be the case that, within a session, a source extension or a compile command for a particular programming language has changed.

- Dump data of environments and configs collection in the nosql container.
- Bring the dump files onto the base machine.
- Parse the data as required by our class. Wrote a PHP script for this too by the name **createStatic.php**.
- The script will create 2 files which will be read by the methods of our class.

### 5.5.1 Reduction in Queries

FIGURE 5.7: Queries executed during compilation after our changes

FIGURE 5.8: Queries executed during execution after our changes

FIGURE 5.9: Queries executed during evaluation after our changes

### 5.5.2 Comparative Results

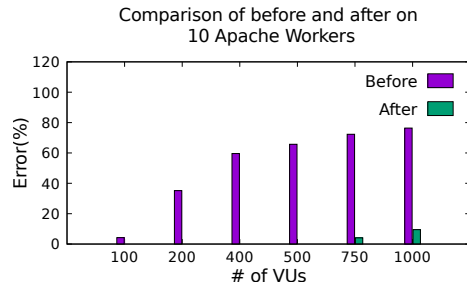


FIGURE 5.10: Comparison of before and after for 10 Apache Workers

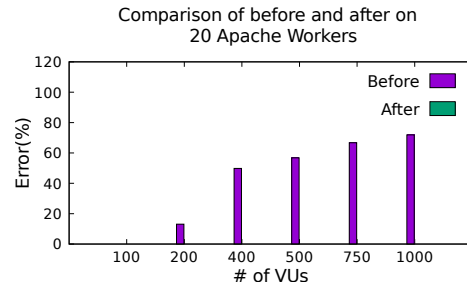


FIGURE 5.11: Comparison of before and after for 20 Apache Workers

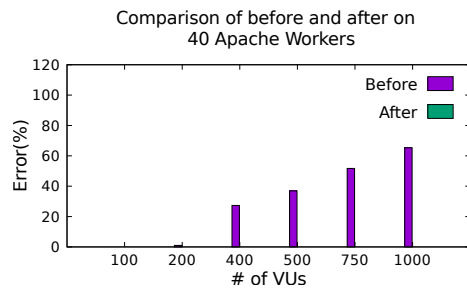


FIGURE 5.12: Comparison of before and after for 4 engines

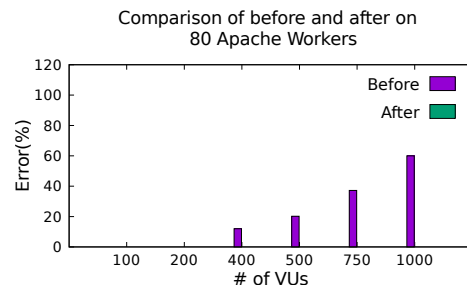


FIGURE 5.13: Comparison of before and after for 8 engines

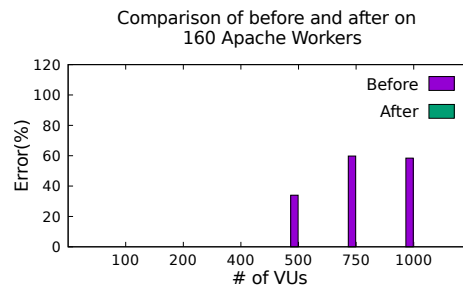


FIGURE 5.14: Comparison of before and after for 16 engines

The figures above show a comparison of performance of Engine before and after our changes. The violet line denotes the baseline and the green line denotes results after our changes. Our changes led the Engine to serve 1000 users with no error% and the average response time of nearly 13 seconds (as mentioned in table 5.11).

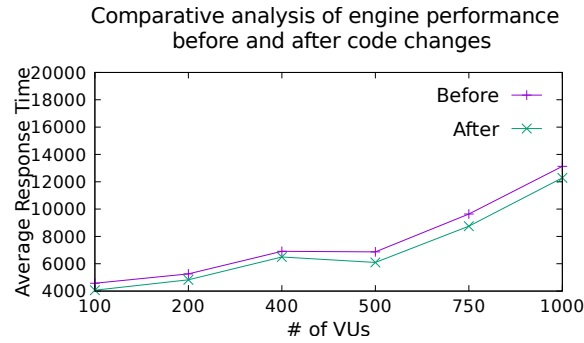


FIGURE 5.15: Comparison of engine before and after our code changes

Here we have shown the comparison between the response times achieved in 2 scenarios. The violet line denotes the scenario just after parameter tuning of Engine. The green line denotes the scenario after all the code changes along with parameter tuning. For a particular number of user, these response times were achieved using the same number of workers and same parameter values. We can see that the green line is consistently below the violet line thereby making the operations faster by nearly 1 second on an average per query.

## Chapter 6

# End Result Performance Comparison

At IITKanpur, Prutor has been deployed on a machine with 32GB RAM and 48 core Xeon 1.87 GHz processor. Whatever results we have gathered, were on a machine with 16GB RAM and 8 core i7 3.40 GHz processor. Therefore our work would be incomplete, if we do not test the performance on the production server. In this chapter we have shown via several plots that the 8 core machine far outperforms the 48 core machine. Just like Sec. 3.4.5 and Sec. 5.5.2, we have shown comparative results for 48 core machine also. This is then followed by the combined results on both machines. It then finally concludes with a tabulated summary of the parameter tuning done throughout the thesis.

### 6.1 Comparative Results of WebApp on 48 core machine

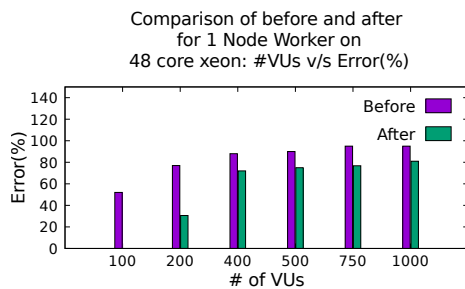


FIGURE 6.1: Comparison of before and after for 1 Node Worker on 48 core machine

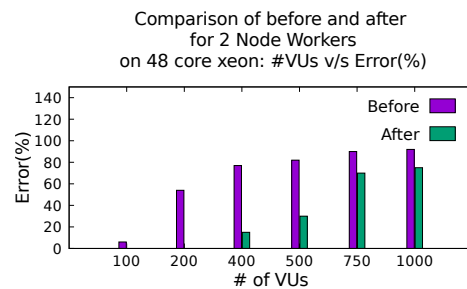


FIGURE 6.2: Comparison of before and after for 2 Node Workers on 48 core machine

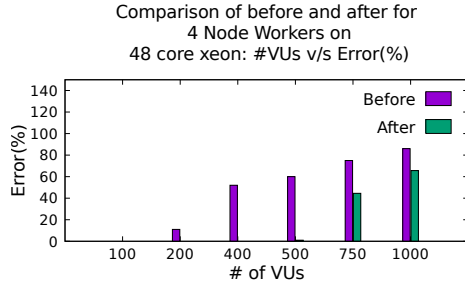


FIGURE 6.3: Comparison of before and after for 4 Node Workers on 48 core machine

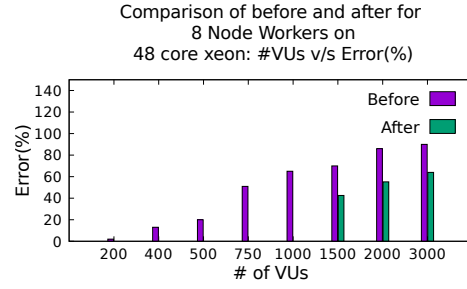


FIGURE 6.4: Comparison of before and after for 8 Node Workers on 48 core machine

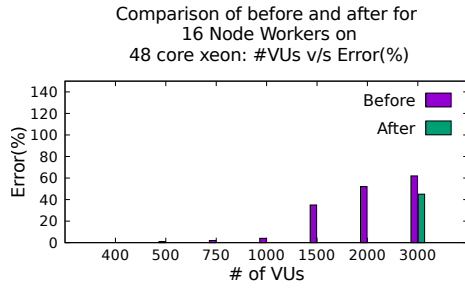


FIGURE 6.5: Comparison of before and after for 16 Node Workers on 48 core machine

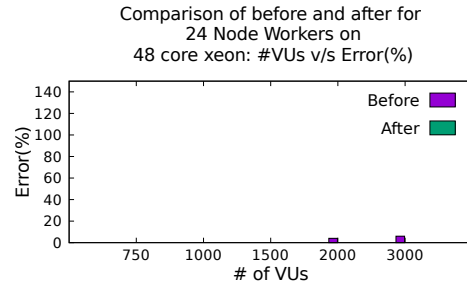


FIGURE 6.6: Comparison of before and after for 24 Node Workers on 48 core machine

## 6.2 Comparative Results of Engine on 48 core machine

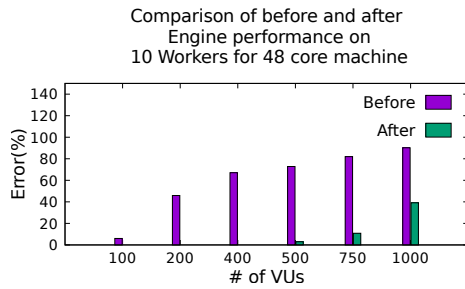


FIGURE 6.7: Comparison of before and after for 10 Apache Worker on 48 core machine

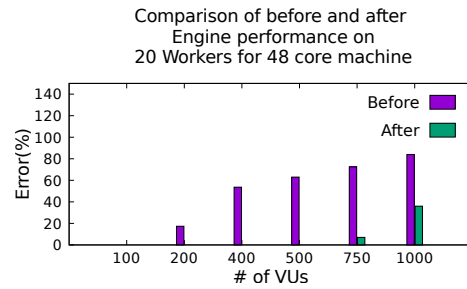


FIGURE 6.8: Comparison of before and after for 20 Apache Workers on 48 core machine

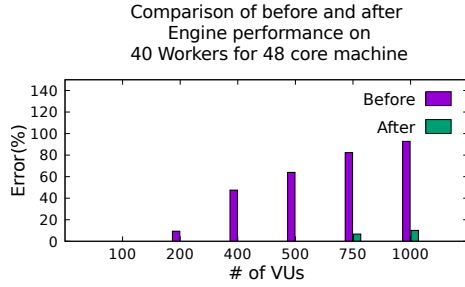


FIGURE 6.9: Comparison of before and after for 40 Apache Workers on 48 core machine

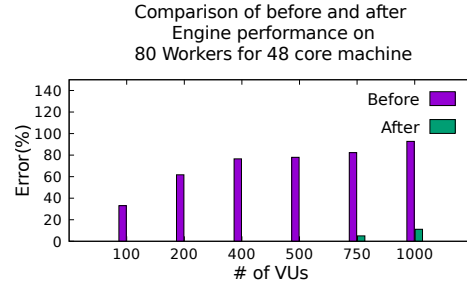


FIGURE 6.10: Comparison of before and after for 80 Apache Workers on 48 core machine

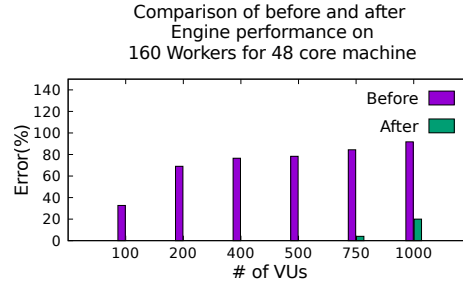


FIGURE 6.11: Comparison of before and after for 160 Apache workers on 48 core machine

### 6.3 8 core i7 V/s 48 core xeon Comparison of WebApp

The plots here show a comparison of the performance of WebApp on both the machines. In the Fig. 6.12, our metric on x axis is number of users and on y axis is number of node workers required. For a particular value of x, we have captured the number of Node workers that were needed to handle x users without any error% and a satisfactory response time.

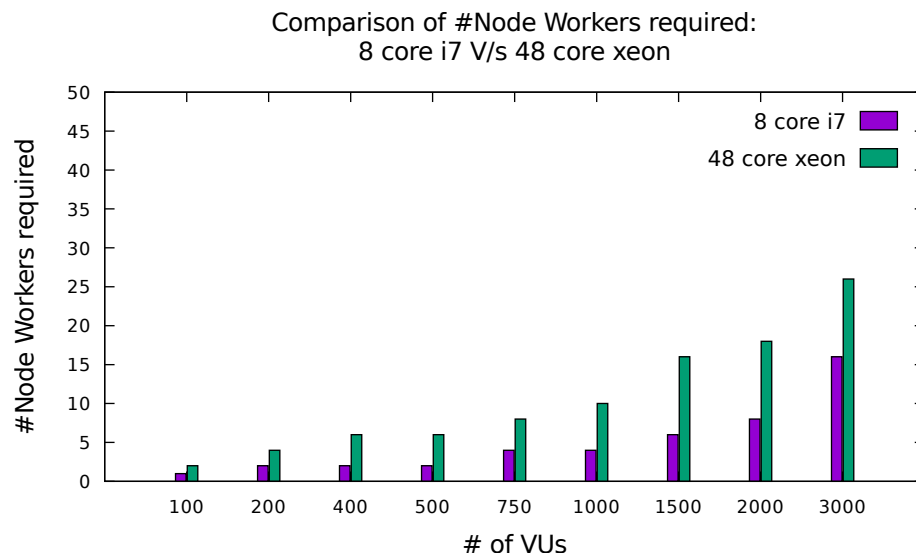


FIGURE 6.12: Comparison of Node workers on 8 core i7 and 48 core xeon

The plot indeed shows that to support the same number of users, the 8 core machine requires far less amount of node workers than the 48 core machine.

In the Fig. 6.13, we have plotted the response times for each x value, achieved with the respective number of node workers shown in the above plot. Now in the case of WebApp as 90% queries are read-only, we got a very satisfactory and reasonable response time in each case. We could have achieved a better response time for each value, but for that we had to increase the number of node workers. This in turn would have meant giving more computing resources to WebApp, even when it is not required.

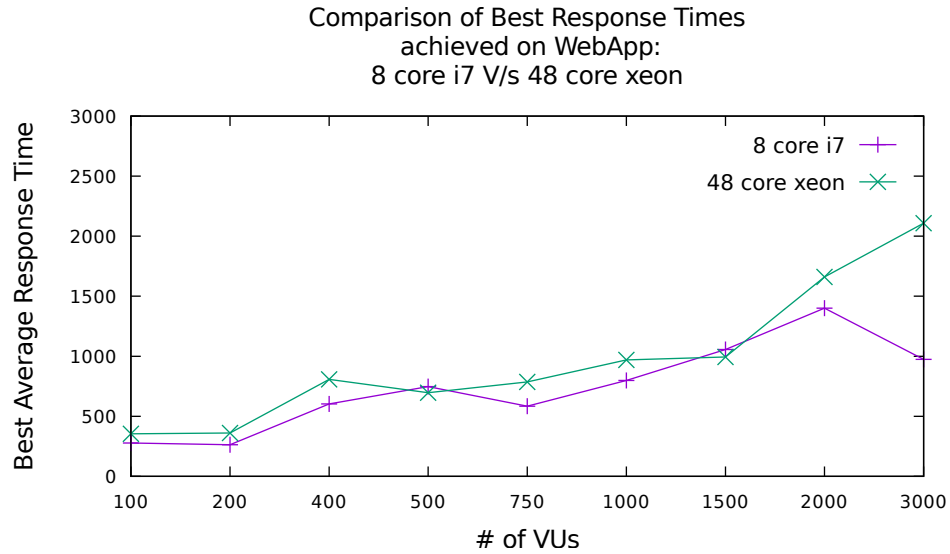


FIGURE 6.13: Comparison of WebApp Response Times on 8 core i7 and 48 core xeon

We can see that the response time of the 8 core machine (violet line) is always lower than the response time of 48 core machine, except for 2 points. We executed the test for 500 users on double the node workers as shown and found the response time nearly halved, which was not even required, so we stayed with this value. In addition to this, if we notice the lines after 2000, we will see that the 48 core line keeps on rising, but the 8 core line drops. The reason for this drop is that we increased the number of node workers for 3000. This shows that in case of 8 core machine, our system has not got saturated yet and it is indeed responding to the horizontal scaling.

## 6.4 8 core i7 V/s 48 core xeon Comparison of Engine

Here for each value of number of users, we have compared the number of Apache workers required to handle their workload with 0 error%. In the case of 48 core machine, for 750 and 1000 users we were not able to get full 100% success. In case of 750 users we got 0.87% and for 1000 we got 6% error%, whereas 8 core machine reported full 100% success with lesser response time and fewer apache workers.

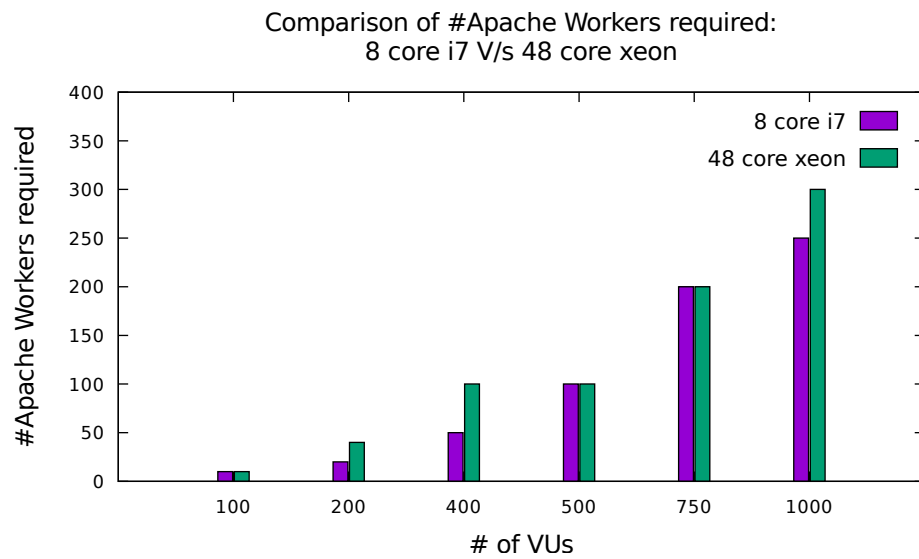


FIGURE 6.14: Comparison of Apache Workers on 8 core i7 and 48 core xeon

The above figure shows that the number of apache workers required by the 8 core machine are always less than or equal to the number reported by the 48 core machine.

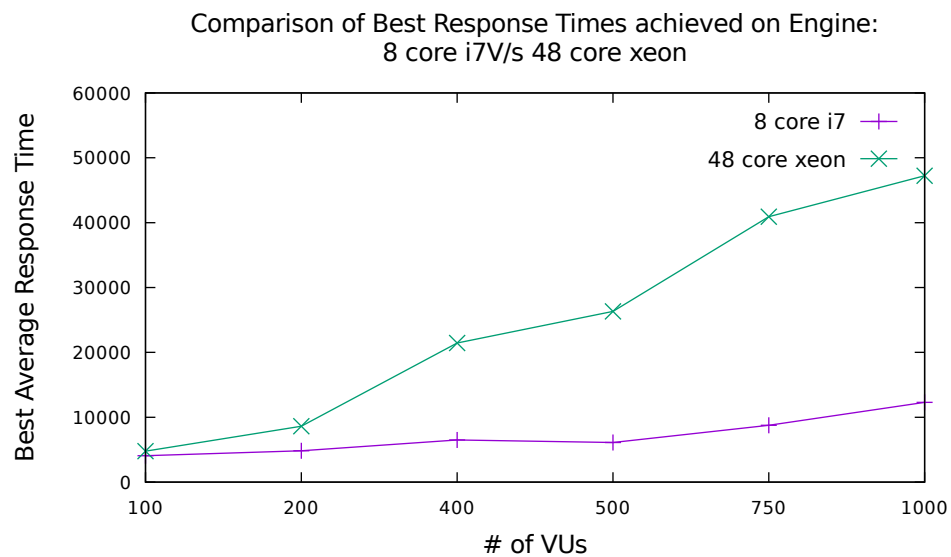


FIGURE 6.15: Comparison of Engine Response Times on 8 core i7 and 48 core xeon

Here the difference is drastic. Where in the most extreme scenario also, the 8 core line barely crosses 10 seconds mark, the 48 core line reaches 20 seconds for half the load and skyrockets to nearly 50 seconds for 1000 users.

We also conclude that the performance of the 8 core i7 is far more superior than the 48 core xeon, in every aspect. Although with 48 core machine we have been able to reach equal number of users as the 8 core i7, but the response times and number of servers required are far more than we need with the 8 core machine.

## 6.5 Variation of WebApp Performance based on cores

Here we study the weightage of cores in webapp performance. To do that we have picked up the scenario of 1000 users. For parameter tuning for 1000 users, we refer to the table 6.4. Now here though the experimentation has been done on the 48 core machine, we have simulated less cores by restricting the core usage of the docker containers of Prutor. For restricting core usage, we have used the hook `-cpustat-cpus` along with `docker run` command.

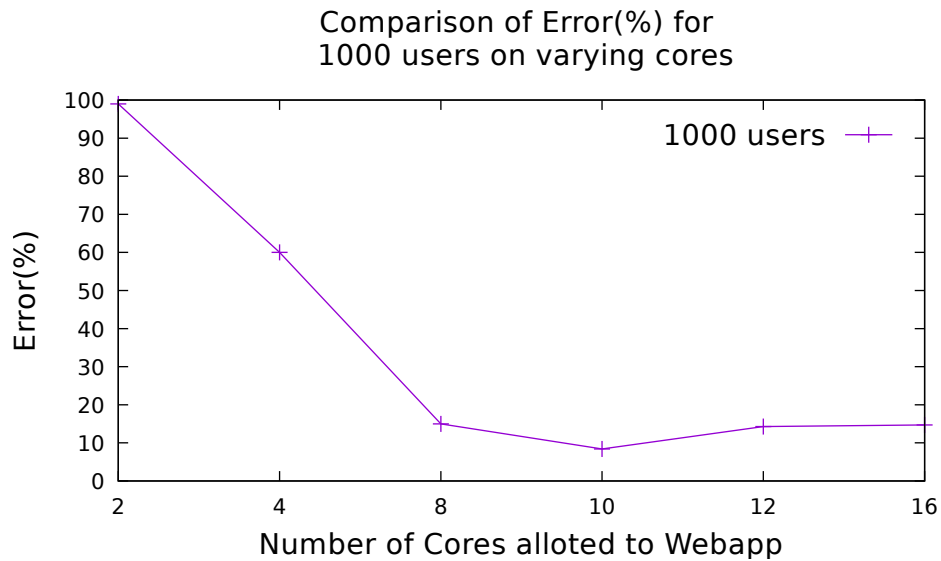


FIGURE 6.16: Comparison of Engine Response Times on 8 core i7 and 48 core xeon

## 6.6 Parameter Summary

Due to lack of column space, some acronyms have been used. They are as follows.

- ESM - Engine Server Maxconn
- LM - Listen Maxconn

- MRW - MaxRequestWorkers
- NW - Node Workers

Here NW Best and MRW Best indicates that if if we increase the respective parameter to the value in these columns, then the response time will be better than satisfactory. Increasing beyond these vaues will be an overkill. If you need to increase beyond these values, then that means, it has nothing to do with the scalability and something else is wrong.

### 6.6.1 Engine on 8 core

Users	ESM	LM	MRW	MRW Best	Node workers
100	100	5000	10	10	1
200	100	5000	20	20	2
400	100	5000	50	50	4
500	100	5000	100	100	4
750	200	5000	200	200	6
1000	200	5000	250	250	6

TABLE 6.1: Parameter values for Engine on 8 core machine

### 6.6.2 WebApp on 8 core

Users	WSM	LM	NW	NW Best
100	100	500	1	1
200	200	1000	2	2
400	400	2000	2	2
500	500	2000	2	2
750	800	3000	4	4
1000	1000	4000	4	4
1500	1500	5000	6	6
2000	2000	5000	8	8
3000	3000	6000	16	16

TABLE 6.2: Parameter values for WebApp on 8 core machine

Although the listen maxconn should be set to a higher value like atleast 5000 for all the cases, but we have anyway given the values that we found to be just as sufficient for the respective case.

### 6.6.3 Engine on 48 core

Users	ESM	LM	MRW	MRW Best	Node workers
100	100	5000	10	10	2
200	100	5000	20	40	4
400	100	5000	50	100	6
500	100	5000	100	100	6
750	200	5000	200	200	8
1000	200	5000	300	300	10

TABLE 6.3: Parameter values for Engine on 48 core machine

### 6.6.4 WebApp on 48 core

Users	WSM	LM	NW	NW Best
100	100	500	1	2
200	200	1000	2	4
400	400	2000	4	6
500	500	2000	4	6
750	800	3000	8	8
1000	1000	4000	8	10
1500	1500	5000	16	16
2000	2000	5000	16	18
3000	3000	6000	24	26

TABLE 6.4: Parameter values for WebApp on 48 core machine

**6.6.5 Combined on 8 core**

Users	WSM	ESM	NW	NW Best	MRW	MRW Best
100	100	100	1	1	10	10
200	100	100	1	1	10	20
400	100	100	2	2	20	50
500	200	100	2	2	50	50
750	400	200	4	4	100	150
1000	400	200	4	4	200	250

TABLE 6.5: Parameter values for Handling Combined Workload on 8 core machine

**6.6.6 Combined on 48 core**

Users	WSM	ESM	NW	NW Best	MRW	MRW Best
100	100	100	1	1	10	10
200	200	100	2	2	20	40
400	400	100	4	4	50	100
500	500	100	4	6	100	150
750	500	200	8	8	150	250
1000	500	200	8	10	250	300

TABLE 6.6: Parameter values for Combined Workload on 48 core machine

## Chapter 7

# Conclusion and Future Work

### 7.1 Conclusion

The main conclusion of our thesis is, if we want our system to be horizontally scalable, we should pay special attention in tuning the components in its technology stack. Simple spawning of more servers will, at some point make the system saturated and resistant to scaling out, which was exactly the case with prutor.

We saw how monitoring tools can give us really meaningful log insights. The Error Analytics feature of NewRelic has been quite instrumental in leading us to the reverse proxy. We witnessed how Database Abstraction can prove to be an overkill in an application. ORM removal reduced the number of queries from 12 to 3 per compilation. We also learnt that API should use database in a judicious manner. By judicious, we mean that if some parameters can be passed as HTTP Request Parameters, we should use that, instead of repeatedly fetching them from the database.

We also ran benchmark tests on 8 core i7 machine and 48 core xeon machine. We finally arrived at a conjecture that 8 core i7 machine due to its superior clock speed (3.40 GHz) far outperforms the 48 core xeon machine (1.87 GHz). We conclude by giving a tabular summary of parameters for each component, which can be used by Prutor admins to prepare servers according to the expected number of users.

## 7.2 Future Work

- Although NewRelic Error analytics and Transaction traces gave a lot of useful information, we still feel that a more in-depth profiling will be of more use in future. For that one can refer linux profiling tools like gperf.
- ORM removal gave us some substantial gain in the performance, but we feel that the whole request and response flow of compilation needs to be re-architected.
- In a complete request cycle of compilation, the engine has to accept the connection, run the code in sandbox, insert compilation results in database, process these results for better readability and then send them to the client as response. Currently, after sending a compilation request, a user is blocked till he/she receives the response. In case of heavy workload, it may lead to wastage of lot of time. This whole model of a transaction is categorized as synchronous or blocking. This can be done in a non-blocking manner with the help of Distributed Job Servers like Gearman. Specifically the job of compilation and inserting the logs into the database can be outsourced to Gearman workers and the job of fetching the results should be left to the client.
- In the current system, every component is running on a single system in docker containers. Some experimentations can be done with a distributed setup, where we can give separate machines to Engine and WebApp.

# Bibliography

- [1] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani, “Prutor: A system for tutoring CS1 and collecting student programs for analysis,” *CoRR*, vol. abs/1608.03828, 2016.
- [2] “Coursera — online courses & credentials from top educators. join for free.” <https://www.coursera.org/>.
- [3] “edx — online courses from the world’s best universities.” <https://www.edx.org/>.
- [4] “Free online courses and nanodegree programs — udacity.” <https://in.udacity.com/>.
- [5] “Get started, part 1: Orientation and setup — docker documentation.” <https://docs.docker.com/get-started/>.
- [6] “What are containers (i.e. docker linux containers / software cont.” <https://www.sdxcentral.com/cloud/containers/definitions/what-are-containers-like-docker-linux-containers/>.
- [7] “Haproxy version 1.7.10 - configuration manual.” <https://cbonte.github.io/haproxy-dconv/1.7/configuration.html>.
- [8] “Benefits of layer 7 load balancing — nginx load balancer.” <https://www.nginx.com/resources/glossary/layer-7-load-balancing/>.
- [9] “Mysql :: Mysql 5.7 reference manual.” <https://dev.mysql.com/doc/refman/5.7/en/>.
- [10] “Mongodb documentation.” <https://docs.mongodb.com/>.
- [11] “Github - memcached/memcached: memcached development tree.” <https://github.com/memcached/memcached>.

- [12] “Node.js express framework.” [https://www.tutorialspoint.com/nodejs/nodejs\\_express\\_framework.htm](https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm).
- [13] “Php: Php manual - manual.” <http://php.net/manual/en/>.
- [14] “Sandbox (computer security) - wikipedia.” [https://en.wikipedia.org/wiki/Sandbox\\_\(computer\\_security\)](https://en.wikipedia.org/wiki/Sandbox_(computer_security)).
- [15] “[https://raw.githubusercontent.com/pixelb/ps\\_mem/master/ps\\_mem.py](https://raw.githubusercontent.com/pixelb/ps_mem/master/ps_mem.py).” [https://raw.githubusercontent.com/pixelb/ps\\_mem/master/ps\\_mem.py](https://raw.githubusercontent.com/pixelb/ps_mem/master/ps_mem.py).
- [16] “Welcome to etd @ iit kanpur: Impact of component selection and configuration parameters on the performance of a web application.” <http://172.28.64.70:8080/jspui/handle/123456789/14415>.
- [17] “Welcome to etd @ iit kanpur: Application level benchmarks: Mms on lamp and mean stacks.” <http://172.28.64.70:8080/jspui/handle/123456789/15460>.
- [18] “Impact of deployment architectures and virtualisation platforms on the performance of a web application.” <http://172.28.64.70:8080/jspui/bitstream/123456789/14414/2/12111053.pdf>. (Accessed on 06/16/2018).
- [19] “Welcome to etd @ iit kanpur: Application level benchmarks: Mooc management system in erlang and node.js.” <http://172.28.64.70:8080/jspui/handle/123456789/16272>. (Accessed on 06/16/2018).
- [20] “Y8111020.pdf.” <http://172.28.64.70:8080/jspui/bitstream/123456789/11748/6/Y8111020.pdf>. (Accessed on 06/16/2018).
- [21] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, “Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR ’16, (New York, NY, USA), pp. 1–12, ACM, 2016.
- [22] “Performance testing vs. load testing vs. stress testing — blazemeter.” <https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing>.
- [23] “System under test - wikipedia.” [https://en.wikipedia.org/wiki/System\\_under\\_test](https://en.wikipedia.org/wiki/System_under_test).

- [24] “What is performance testing and types of performance testing?.” <http://www.softwaretestingclass.com/what-is-performance-testing/>.
- [25] “Load testing software: Application testing tools.” <https://software.microfocus.com/en-us/products/loadrunner-load-testing/overview>.
- [26] “Webload - website and application performance testing.” <https://www.radview.com/webload-download/>.
- [27] “Apache jmeter - user’s manual: Building a web test plan.” <https://jmeter.apache.org/usermanual/build-web-test-plan.html>.
- [28] “Fundamentals of performance profiling.” <https://smartbear.com/learn/code-profiling/fundamentals-of-performance-profiling/>.
- [29] “File system — node.js v10.1.0 documentation.” <https://nodejs.org/api/fs.html>.
- [30] “Node.js cluster and express.” <https://rowanmanning.com/posts/node-cluster-and-express/>.
- [31] “learning-express-cluster/app.js at master · rowanmanning/learning-express-cluster · github.” <https://github.com/rowanmanning/learning-express-cluster/blob/master/app.js>.
- [32] “Increasing the maximum number of tcp/ip connections in linux - stack overflow.” <https://stackoverflow.com/questions/410616/increasing-the-maximum-number-of-tcp-ip-connections-in-linux>.
- [33] “https://klaver.it/linux/sysctl.conf.” <https://klaver.it/linux/sysctl.conf>.
- [34] “Tuning your linux kernel and haproxy instance for high loads.” <https://medium.com/@pawilon/tuning-your-linux-kernel-and-haproxy-instance-for-high-loads-1a2105ea553e>.
- [35] “Mysql :: Mysql 5.7 reference manual :: 8.8 understanding the query execution plan.” <https://dev.mysql.com/doc/refman/5.7/en/execution-plan-information.html>.
- [36] “Materialized views with mysql — mysql, galera cluster and mariadb support and services.” <http://www.fromdual.com/mysql-materialized-views>.
- [37] “Getting started with mysql stored procedures.” <http://www.mysqltutorial.org/getting-started-with-mysql-stored-procedures.aspx>.

- [38] "Mysql triggers." <http://www.mysqltutorial.org/mysql-triggers.aspx>.
- [39] "innodb.buffer\_pool\_size: get the best of your memory - speedemy." [https://www.speedemy.com/mysql/17-key-mysql-config-file-settings/innodb\\_buffer\\_pool\\_size/](https://www.speedemy.com/mysql/17-key-mysql-config-file-settings/innodb_buffer_pool_size/).
- [40] "Calculating innodb buffer pool size for your mysql server." <https://scalegrid.io/blog/calculating-innodb-buffer-pool-size-for-your-mysql-server/>.
- [41] "Mysql innodb performance improvement: Innodb buffer pool instances - updated! - sysadmins of the north." <https://www.saotn.org/mysql-innodb-performance-improvement/>.
- [42] "Monitoring haproxy performance metrics." <https://www.datadoghq.com/blog/monitoring-haproxy-performance-metrics/>.
- [43] "Redbeanphp :: Welcome." <https://redbeanphp.com/index.php>.
- [44] "Object-relational mapping - wikipedia." [https://en.wikipedia.org/wiki/Object-relational\\_mapping](https://en.wikipedia.org/wiki/Object-relational_mapping).
- [45] "Php prepared statements." [https://www.w3schools.com/php/php\\_mysql\\_prepared\\_statements.asp](https://www.w3schools.com/php/php_mysql_prepared_statements.asp).
- [46] "(the only proper) pdo tutorial - treating php delusions." <https://phpdelusions.net/pdo>.
- [47] "How to install java on ubuntu 16.04 — rosehosting blog." <https://www.rosehosting.com/blog/how-to-install-java-on-ubuntu-16-04/>. (Accessed on 06/01/2018).
- [48] "amazon web services - setting up jmeter for distributed testing in aws with connectivity issues - stack overflow." <https://stackoverflow.com/questions/16618915/setting-up-jmeter-for-distributed-testing-in-aws-with-connectivity-issues/32260139#32260139>.
- [49] "Mysql :: A quick guide to using the mysql apt repository." <https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/>.
- [50] "Install servers for linux with yum or apt — new relic documentation." <https://docs.newrelic.com/docs/servers/new-relic-servers-linux/installation-configuration/install-servers-linux-yum-or-apt>.

# Appendices

## Appendix A

# Apache JMeter: Setup Instructions

### A.1 Apache JMeter: Setup Guide

#### A.1.1 Java Installation

These steps have been taken from [\[47\]](#).

- `sudo apt-get update && sudo apt-get -y upgrade`
- `sudo apt-get install software-properties-common`
- `sudo apt-add-repository ppa:webupd8team/java`
- `sudo apt-get update`
- `sudo apt install oracle-java8-installer`
- `java -version`

#### A.1.2 JMeter Installation

- **If GUI available** Go to the downloads page of Apache JMeter. Look for the latest release of Apache JMeter. In this thesis we used Apache-JMeter 3.3, which requires Java 8 or Java 9. Download it.

- **If GUI not available** Run the following command: `wget www-us.apache.org/dist/jmeter/binaries/apache-jmeter-3.3.tgz`
- Place the JMeter archive in your home directory. Any directory will work.
- `cd home`
- `tar -xvzf apache-jmeter-3.3.tgz`
- This will create directory `apache-jmeter-3.3`
- `cd apache-jmeter-3.3`
- `cd bin`
- Inorder to launch the GUI run `./jmeter`
- Inorder to launch it in GUI mode with a test plan run `./jmeter -t <test_plan>.jmx`

**Test plans in JMeter when viewed in any editor have xml like structure and have extension as .jmx**

**Here <test\_plan >, path relative to bin/jmeter binary or the absolute path from root will work.**

By following the above steps, you will be able to setup JMeter on your machine. When JMeter is run with a sample test plan, following is the screen that you will see.

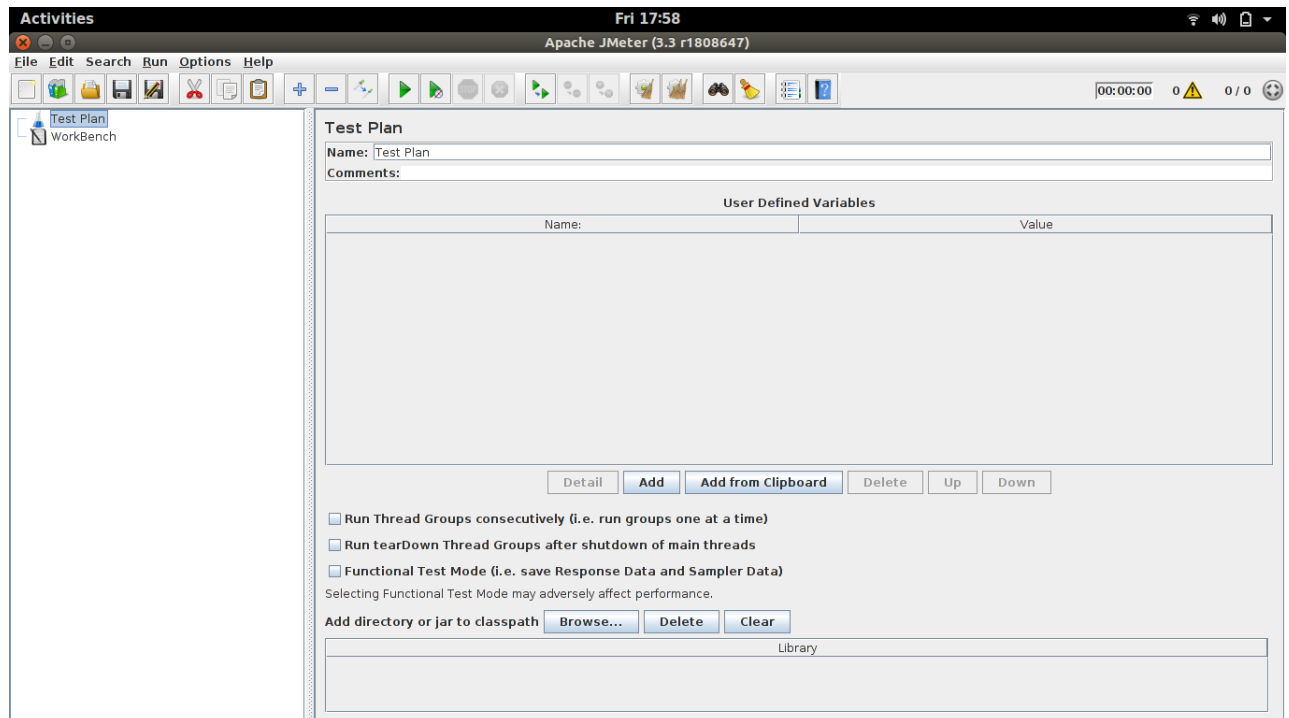


FIGURE A.1: JMeter:Home Screen

### A.1.2.1 Jargons of JMeter

This screen depicts a typical view of JMeter in GUI mode. Let's go through the important functionalities and Jargons that it offer one by one.

- **Test Plan** - A Test Plan typically describes the steps that Jmeter will execute when run. It comprises of a huge number of elements. For a detailed view of the configurations related to a test plan, please visit the link <http://jmeter.apache.org/usermanual/build-test-plan.html>.
- **Thread Group** - The thread group element controls the number of threads, JMeter will generate during a run. The controls for a thread group allow you to do the following.
  - Set the number of threads
  - Set the ramp-up period
  - Set the number of times to execute the test

### Adding Thread Group to a test plan

- Right click on Test Plan.
- Select Add → Threads(users)→ ThreadGroup.

After following the above steps you will see the following screen.

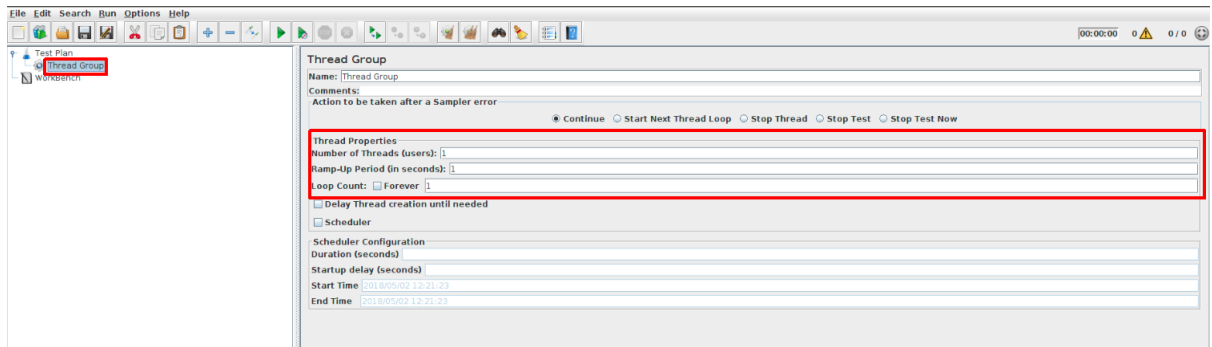


FIGURE A.2: JMeter:Thread Group added

- **Number of Threads** - Each thread will execute the complete test plan independent of other test threads. Multiple threads are used to simulate concurrent connections to your server application.
- **Ramp-up period** - The ramp-up period tells JMeter how long to take to "ramp-up" to the full number of threads chosen. If 10 threads are used, and the ramp-up period is 100 seconds, then JMeter will take 100 seconds to get all 10 threads up and running. Each thread will start 10 (100/10) seconds after the previous thread was begun. If there are 30 threads and a ramp-up period of 120 seconds, then each successive thread will be delayed by 4 seconds.  
Ramp-up needs to be long enough to avoid too large a work-load at the start of a test, and short enough that the last threads start running before the first ones finish (unless one wants that to happen).

- **Loop count** - The loop count tells, how many times each thread will run.

Now whatever elements are needed will be added to the Thread Group.

- **Controllers** - JMeter has 2 types of controllers.
  - **Samplers** - Samplers tell JMeter to send requests to a server and wait for a response. They are processed in the order they appear in the tree. Some of the key JMeter samplers include.

- \* HTTP Request.
- \* JDBC Request.
- \* FTP Request.

Each Sampler has its own options and can be configured to suit user's Requirements. We have only used HTTP Requests, so we here will explain configuration of HTTP Request sampler.

- **Logic Controllers** - Through Logic Controllers, Jmeter decide, the order and number of times to send the request. The only Logic controller that I have used is **Only-Once controller**.

**Only-Once controller**- All the samplers under this controller are only executed once by each thread. It is generally used with HTTP request for login.

- **Listeners** - Listeners provide a tabular or graphical representation of the post-run metrics calculated/collected by JMeter.

### A.1.3 Step-By-Step Construction of a test plan

Now we will demonstrate, how to use these elements in forming a test plan. The name of the test plan is **Prutor\_webapp.jmx**. At the end it should appear like this.

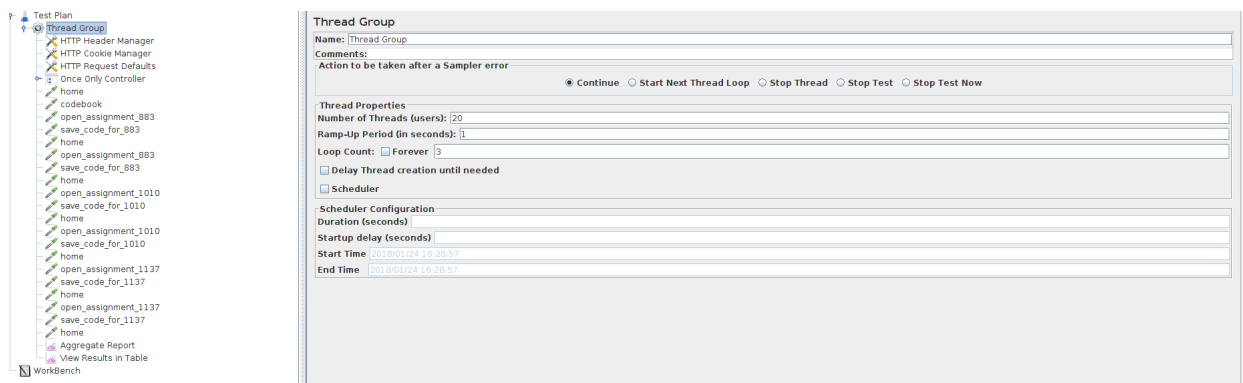


FIGURE A.3: JMeter:Final Test Plan

Elements used are.

- Samplers
  - HTTP Requests
    - \* home

- \* codebook
  - \* open\_assignment
  - \* save\_code
- Listeners
  - Aggregate Report
  - View Results in a tree
- Logic Controllers
  - Only-once controller

## Steps

- Launch Jmeter in GUI mode.
- Right-click on the Test Plan in left panel
- Navigate as Add → Threads(users) →
- Now, you can change the number of users and ramp-up period in the right panel as marked.

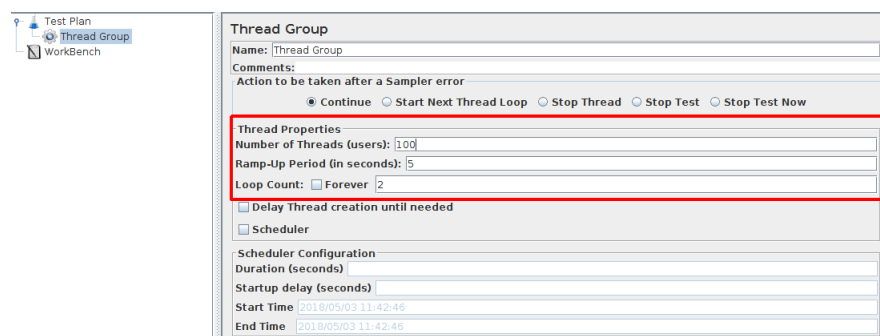


FIGURE A.4: JMeter:Thread Group properties

The configuration in the fig. A.4 means, that this test plan will run with 100 threads simulating 100 users, spawned over a period of 5 secs. In more layman terms, it will take 5 secs to completely reach the load of 100 concurrent users.

**Adding the HTTP Requests** The description of these HTTP requests can be found in sec. 3.1.1 in chap. 3. Here we have described the configuration parameters that were set.

- **Login -**
  - Adding **Once-Only** controller.
    - \* Right-click on the Thread Group.
    - \* Navigate as Add→Logic Controllers→Once Only Controller.
  - Adding **HTTP Request** sampler.
    - \* Right-click on the Thread Group.
    - \* Navigate as: Add→Sampler→HTTP Request.
  - Filling up the configurations for Login Request. In the right panel, in the Basic Tab. No need of Advanced Tab.
    - \* **Method:** POST

- \* **Server Name or IP:** 172.27.20.29 (The ip of the machine hosting Prutor)
- \* **Port Number:** 82  
Port Number on which container named webproxy is running. This is because the webproxy (HAProxy inside) is the first container which our request hits.
- Adding the username and password.
  - \* Goto the Parameters (default) tab.
  - \* Click on Add.
  - \* Type username in the Name field.
  - \* Type user3@gmail.com or any other username.
  - \* Click on Add.
  - \* Type password in the Name field.
  - \* Type escits or the password for your respective username.

The output will be as follows.

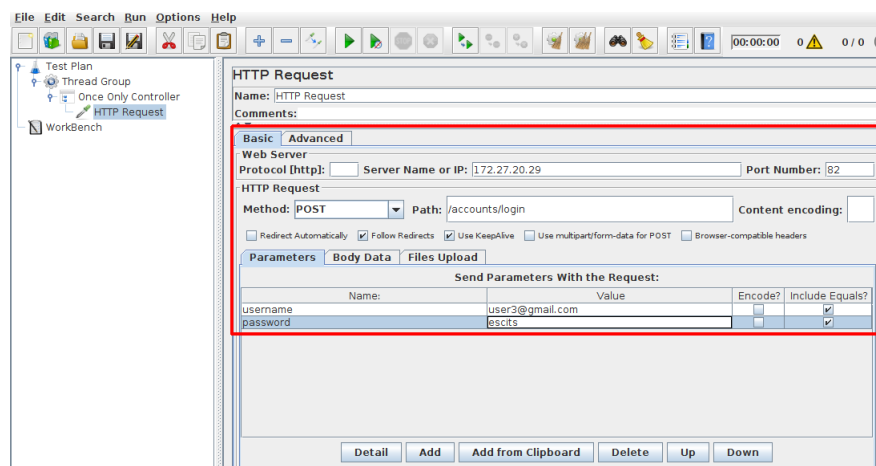


FIGURE A.5: JMeter: Login Request Added

- **home-**

- Adding the HTTP Request sampler.
- Fill in Server Name, Port, Method as explained above.
- Path: /home
- Name: home **For home there are no GET parameters.**

The test plan so far will look as.

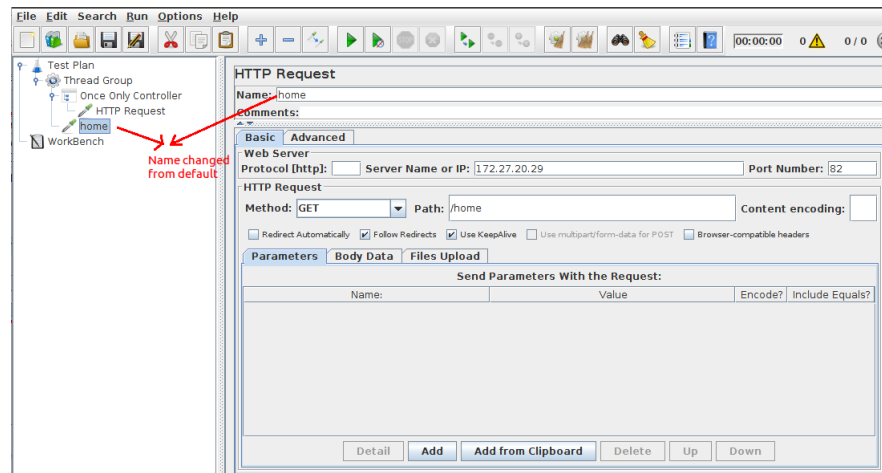


FIGURE A.6: JMeter:Home Request Element

- **codebook** -

- Adding the HTTP Request sampler.
- Fill in Server Name, Port, Method as explained above.
- Path: /codebook
- Name: codebook

- **open\_assignment**

- Adding the HTTP Request sampler.
- Fill in Server Name, Port, Method as explained above.
- Path: /editor/<assignment\_id >.
- Name: open\_assignment.

- **save\_code**

- Adding the HTTP Request sampler.
- Fill in Server Name, Port, Method as explained above.
- **Method:** POST
- **Name:** save\_code
- **Parameters**
  - \* assignment\_id
  - \* branch\_id

\* code: **url encoded code**

\* Trigger: manual/submit

Trigger value as manual implies the request is for manually saving the code.

Trigger value submit implies the request is for code submission.

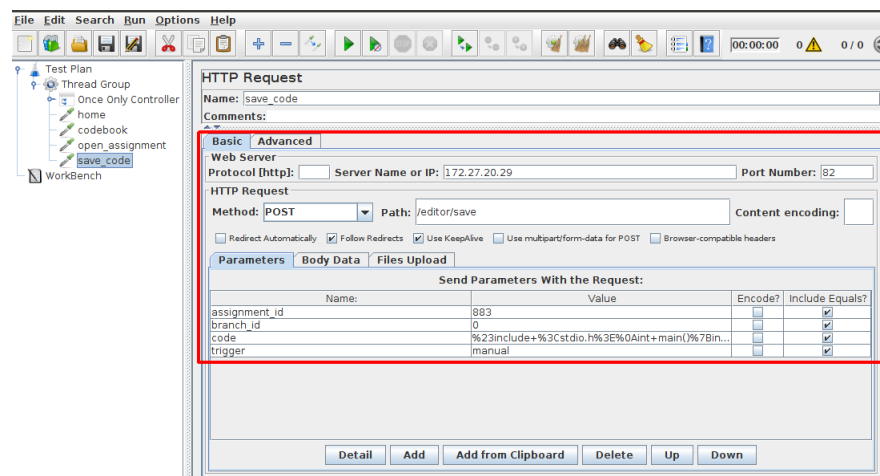


FIGURE A.7: JMeter:Save Code Request

- **Duplicate the requests** Now, you just copy and paste these request samplers.
  - Copy: Right-click on the request sampler in the test tree that you want to be copied and click copy.
  - Paste:- Right-click on the Thread Group in the test tree and click paste.
  - Position it, by dragging the pasted element in whichever order you require.

This will give you the required test plan as shown in fig. A.3.

## A.2 Distributed Load Testing using JMeter

When we want to generate load for very large number of users (>1k), we cannot do it with 1 machine, even though our system has enough system and memory. We use multiple systems in **Master-Slave** configuration to solve the purpose. This configuration allows a single system, running JMeter, to control multiple JMeter machines. The single controller is known as the **Master-client** and other machines are known as **Slave-servers**. We do not need to put the test plan separately on each slave machine, the Master JMeter sends it to its slaves. The test plan is then replicated concurrently for all the slaves, resulting in a large amount of concurrent load.

Before we move on to the setup let us look into some of the **prerequisites** for this setup.

- Download and install Apache JMeter on all the machines.
- Make sure all the machines have the same version of Apache JMeter.
- Make sure all the machines are in same subnet.
- Make sure the version of Java is same on all the machines.
- Disable all the firewalls.

### A.2.1 Distributed Test Setup: Step-by-Step

After ensuring the above prerequisites, please follow the following steps.

- **Set Remote Hosts** This step tells your Master JMeter, which machine(s) it has to control.
  - `cd apache-jmeter-3.3/bin`
  - `vi jmeter.properties`
  - Search for **remote\_hosts**. By default, you will find `remote_hosts=127.0.0.1`.
  - Comment this line out.
  - Write the following line instead: `remote_hosts=<slave_ip1>,<slave_ip2>...<slave_ip_n>`. If in a private network like IITK, these slave ips should be private IPs. For eg:- `remote_hosts=172.27.21.16,172.27.21.17,172.27.21.18`

**Note:** As per most of the blogs and JMeter's official website, this is the only step which is required for the setup. As per these sources, the only step that remains after this is running both the master and slaves by.

– **Slaves**

- \* Goto all your slave-machines one by one.
- \* `cd apache-jmeter-3.3/bin`
- \* Run `./jmeter-server`

– **Master**

- \* `cd apache-jmeter-3.3/bin`
- \* Run `./jmeter`

**But after following these steps, the test plan will not work.**

Follow the top answer in [48] to get the test plan working.

- Additional config to be done on **slaves** taken from the above link.

- Goto each of your slave machine.
- `cd apache-jmeter-3.3/bin`
- `vi jmeter-server`
- You will find a commented line `RMI_HOST_DEF=-Djava.rmi.server.hostname=xxx.xxx.xxx.xxx`
- Change it to `RMI_HOST_DEF=-Djava.rmi.server.hostname=<slaveIP>` For eg: slave IP is 172.27.27.21, it will become.  
`RMI_HOST_DEF=-Djava.rmi.server.hostname=172.27.21.21`
- `vi jmeter.properties`
- Search for `server.rmi.localport`
- Set `server.rmi.localport=4001`. Any non-reserved port should work.
- Launch jmeter-server using the following command `./jmeter-server`

- Additional config to be done on **Master** taken from the above link.

- `cd apache-jmeter-3.3/bin`
- `vi jmeter.properties`
- Search for `client.rmi.localport`

- Set *client.rmi.localport=4000*. Again, any non-reserved port should work.
- Launch JMeter Master client by running the command  
*./jmeter -t <test\_plan.jmx> -Djava.rmi.server.hostname=<masterIP>*

## Appendix B

# MySQL-WorkBench: Setup Instructions

### B.1 Build an Image of MySQL 5.7

- `cd /prutor`
- `mkdir rdbupdated`
- `cp rdb rdbupdated`
- `cd rdbupdated`
- `vi mysql.list`
  - `deb http://repo.mysql.com/apt/ubuntu/trustymysql-5.7`
- `vi Dockerfile`
  - Add this in COPY section.
    - \* `COPY mysql.list /etc/apt/sources.list.d/mysql.list`
  - Add this in install required tools (added `-force-yes` in that line).
    - \* `RUN apt-get update && apt-get install -y -force-yes mysql-server supervisor apache2 php5 php5-mysql php5-mcrypt vim`
  - Change the Grant line User with all accesses section.
    - \* `GRANT ALL ON *.* TO 'prutor'@'%'`. *Earlier it was granting on its.\*, so it resulted in workbench not able to access performance\_schema.*

- In the Allow listening outside section change the file name.
  - \* Replace as: `/etc/mysql/my.cnf` → `/etc/mysql/mysql.conf.d/mysql.conf`. *In 5.5 it was `/etc/mysql/my.cnf`, but in 5.7 it was different.*
- Run the `/prutor/deploy_db` script.

After following all these steps, you will have a container running MySQL 5.7.

## B.2 Setting up MySQL-Workbench

The setup of MySQL-Workbench consists of 2 parts. Each part has been dedicated a subsection.

### B.2.1 Enabling SSH on `rdbupdated` container

We need some mode of connection between the MySQL-workbench application and the database container (**rdb** for MySQL 5.5 and **rdbupdated** for MySQL 5.7). We preferred SSH session between them. In order to be able to do that our database container should be running ssh service. Following are the steps to do that.

- `apt-get update`
- `apt-get install openssh-server`
- `mkdir -p /root/.ssh`
- `chmod 700 /root/.ssh`
- `touch /root/.ssh/authorized_keys`
- `chmod 600 /root/.ssh/authorized_keys`
- `vi /root/.ssh/authorized_keys`
  - Copy your(on which mysql workbench will be running) `/root/.ssh/id_rsa.pub` here.
- `service ssh restart`

### B.2.2 Installing mysql-workbench

- Follow 1 of [49].
- apt-get update
- apt-get install mysql-workbench-community

In order to Launch run `/usr/bin/mysql-workbench`.

## Appendix C

# New-Relic: Setup Instructions

New-Relic offers a variety of products to visualize the performance metrics of your system. Some of them are New-Relic Servers, New-Relic Infrastructure, New-Relic APM. We used New-Relic Servers as it sufficed our requirements. Before beginning the installation, Create an account on <https://newrelic.com/>. Now the installation has 2 parts. First part is the installation of New-Relic servers. It is the New-Relic Servers service which will send out metrics to the New-Relic. The second part is the installation of New-Relic technology specific agent. The role of this agent is to collect the performance metrics from our application server and send it to the New-Relic Servers.

### C.1 New-Relic Servers for Linux

The steps have been taken from [50]. The steps could have easily be read from the mentioned link, but at the time of writing this thesis, New-Relic was in process of removing support for New-Relic Servers. So clicking on the URL kept re-directing to New-Relic Infrastructure page.

- `echo 'deb http://apt.newrelic.com/debian/ newrelic non-free' — sudo tee /etc/apt/sources.list.d/newrelic.list.`

This command adds the repository from which apt will pull newrelic binary.

- `wget -O- https://download.newrelic.com/548C16BF.gpg — sudo apt-key add -`

This step is not mandatory. It will authenticate the source from apt will pull the newrelic binary.

- *apt-get update*
- *apt-get install newrelic-sysmond*
- *nrsysmond-config --set license\_key=YOUR\_LICENSE\_KEY*. You can find your license key in your account settings on the New-Relic website. You can also set license key in */etc/newrelic/nrsysmond.cfg*.
- */etc/init.d/newrelic-sysmond start*

## C.2 Enable Monitoring for Docker

As the application server is running inside a docker-container, so we need to enable newrelic so that it can do performance monitoring inside docker.

- *service stop newrelic-sysmond*
- *service stop docker*
- *usermod -a -G docker newrelic*
- *service start docker*
- *service start newrelic-sysmond*

## C.3 Installing New-Relic Node.js agent

We need only Node.js agent as we did profiling for only the webapp component. So here are the steps.

- *npm install newrelic --save*
- Copy *node\_modules/newrelic.js* to the root directory of your webapp, i.e. at the same level of *app.js*.
- Edit **license key** and **app name** in *newrelic.js*.
- Add *require('newrelic')* in the beginning of *app.js*.
- Ensure *@newrelic/native-metrics* package is installed, if not then do it.

## Appendix D

# Instructions for Parameter Tuning

### D.1 Tuning Engine Parameters

The Engine Parameters that we tuned can be found in `mpm_prefork.conf` file inside the engine container. Prefork is the multiprocessing module of apache. Those parameters are as follows.

- `MaxRequestWorkers`
- `MaxConnectionsPerChild`
- `ServerLimit`

Here are the steps to tune them.

- Run `docker exec -it engine1 bash`
- Open `/etc/apache2/mods-enabled/mpm_prefork.conf`
- Change the values of the parameter you want to change.
- `service apache2 restart`

## D.2 Tuning WebApp Parameters

The only parameter that we tuned inside webapp was the number of node workers. We can also increase the number of node workers by increasing the number of webapp containers. For tuning it from inside, follow these steps.

- Run `docker exec -it webapp1 bash`
- Open `/var/www/bin/www`
- Increase the number of iterations of the for loop.

## D.3 Tuning HAProxy

The Proxy parameters that we tuned were server maxconn, listen maxconn and contimeout. Here are the steps to tune them

- `docker exec -it webproxy bash`
- For listen maxconn
  - Open `/tmp/haproxy.default`.
  - Change the value after maxconn.
- For server maxconn.
  - Open `/root/updater.php`
  - Change the maxconn values of whichever server component you want to change.
- For contimeout
  - Open `/etc/haproxy/haproxy.tmpl`
  - Change the value.
- `supervisorctl stop all`
- `supervisorctl reload`
- `service haproxy restart`
- Verify the values have updated in `/etc/haproxy/haproxy.cfg`

## Appendix E

# Repository links

Here are the git clone urls of the repositories.

### E.1 Prutor

`https://jindalakshay@bitbucket.org/jindalakshay/prutor\_new.git`

### E.2 WebApp

`https://jindalakshay@bitbucket.org/jindalakshay/its-web-new.git`

#### E.2.1 Steps to update Nodejs

`https://docs.google.com/document/d/1shWIbUWm3cr80WSWNLfmG-Qct04s8YKRSr2mr4\_8T64/edit?usp=sharing`

### E.3 Engine

`https://jindalakshay@bitbucket.org/jindalakshay/its-engine-new.git`

### E.3.1 Related to Sec. 5.4.3

- Added StaticConfig.php in /codebase\_new/engine/systems
  - It is the file which contains the class that reads from JSON files.
- Added createStatic.php in /codebase\_new/engine/
  - It is the script which creates the json files that are used by StaticConfig.php.

## E.4 Steps to Setup

- Follow the procedure to deploy prutor as mentioned in the Excel sheet as mentioned in the **Copy of Prutor\_Work** excel sheet in the folder as mentioned in Sec. E.5.
- After deploying everything, follow the below mentioned procedure, as and when required.
- **Setup MVs**
  - To be done on the base machine.
    - \* `cd /prutor_new/rdbupdated`
    - \* Make sure there are some entries in the tables from which the Materialized Views are created.
    - \* `./create_ongoingmv`
- **Setup JSON files for data in MongoDB**
  - `cd /codebase_new/engine`
  - `./dump_nosql`

## E.5 Link of all the important docs related to Thesis

<https://drive.google.com/drive/u/0/folders/1anHThe5EVSZquaJ3Xa2sJUip6XMx6PsI?ths=true>

## E.6 All the JMeter Test Plans

Clone url - `https://jindalakshay@bitbucket.org/jindalakshay/jmeter_test_plans.git`