Report

Room Booking System

(Bhagyesh Patil 16305R003, Ayush Goyal 16305R011)

Phase 3:Optimizing System Performance

New design and optimizations:

In the earlier design of our project, all the data about the rooms booked (Room number, name, aadhaar number, date of booking) was stored in DB using mysql. As mysql is not a in-memory DB (by default), it takes a lot of time in reading and writing from disk which reduces our throughput and increases our response time.

In our new design, we have used **Redis** which is a in-memory DB. Redis stores the data in key-value pair. All the data is stored in RAM until we flush it to the drive. As the data has to be written and read from RAM, the user requests are fulfilled faster as compared to case with mysql. So in case of redis we get high throughput and low response time.

We have used redis hashes to store each tuple where room number is the key and with each field (Name, aadhaar number, etc) the corresponding values are associated. E.g. hgetall 1 (where 1 is Room no) will give:

1) "name"
2) "bhagyesh"
3) "addhar"
4) "652365236523"
5) "date"
6) "12-12-2017"
7) "roomno"
8) "1"

Also the mysql can be made in-memory by writing **ENGINE=MEMORY** while creating the table. We tried this, and it also worked and our throughput was increased in this case.

Load Generator:

The load generator we have implemented is a Closed loop load generator. The load is generated by using a multi-threaded generator which repeatedly send a specified request to the server. Each of the thread act as a client to the server and does not have any interactivity, it just bombards requests to the server in automated fashion.

The load generator can be used to perform load testing of:

1. Book requests only: In this we bombard our server with only book requests and calculate the average throughput and average response time.

2. Mixture of Book and Cancel requests: In this we bombard our server with appropriate percentage of book requests and cancel requests, then we calculate the average throughput and average response time.(Typically 50-80% book and 20-50% cancel. It can be changed in load generator's code.)

We have taken care so that the load generator effectively produces load and does not become the bottleneck. Also we have run the server and load generator on separate machines for our readings.

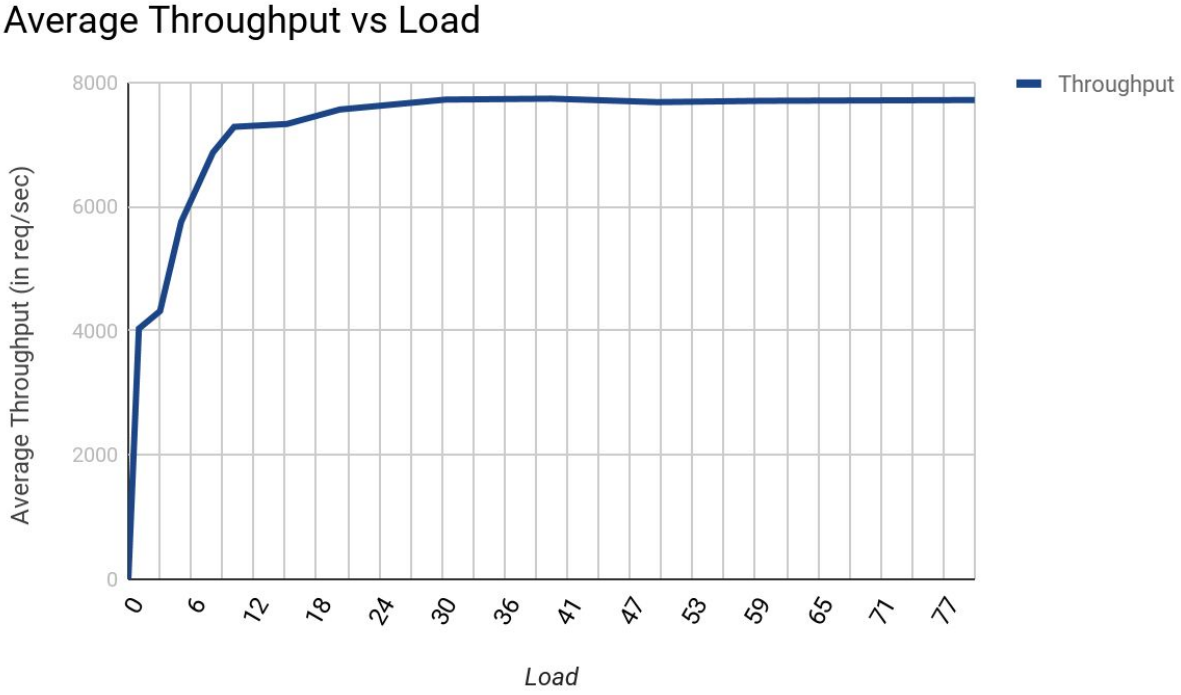
For comparison with phase 2 results, the experiment is made to run for 4 minutes for a given value of load, and the following parameters are measured of our system in steady state : the average throughput of your server (the number of requests/sec being successfully completed, averaged over the entire experiment) and the response time of the server (the average time taken for a request to complete, averaged over all completed requests). These values are measured in the load generator itself and are displayed on the terminal after load test is completed.

The readings of throughput and response time are taken by running the server on different machine. The machine had only two cores enabled (similar to Phase 2) and all other cores were disabled by running a script (set_cpus.sh).

Graphs:

1) When load is only Book Requests:

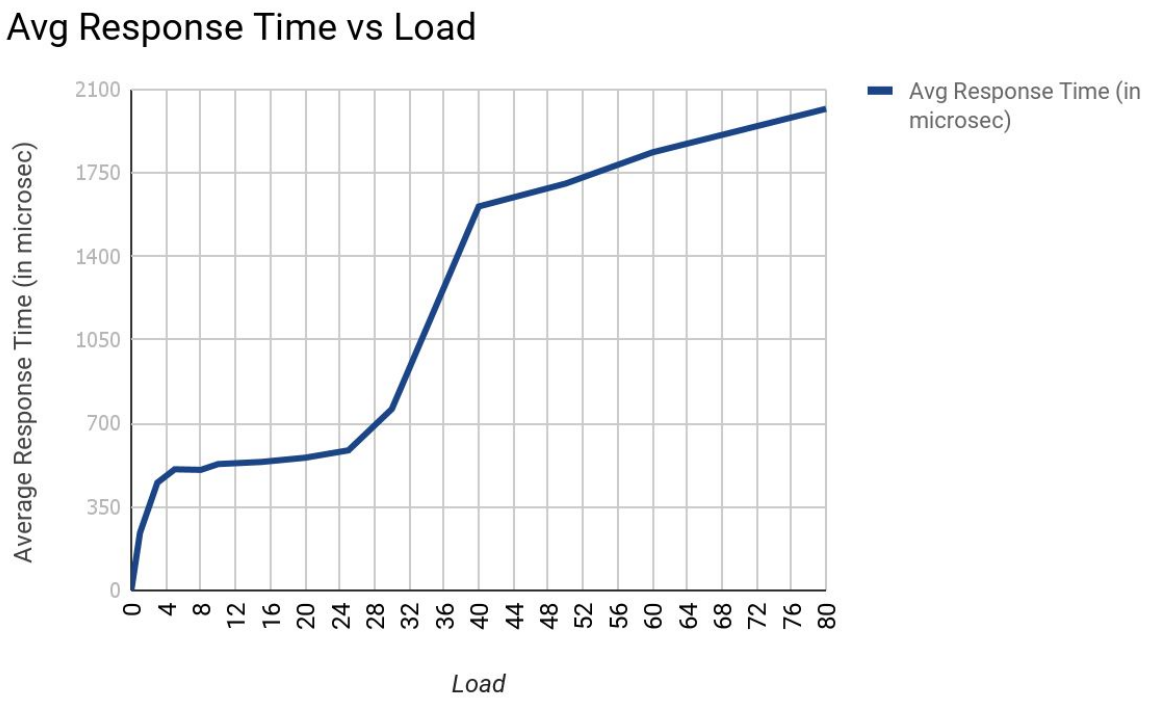
Average throughput vs Load:



Values noted during the experiment on which the above graph is plotted:

	A	B	C
1	Load	Throughput	
2	0	0	
3	1	4036.7	
4	3	4318.983	
5	5	5757.366	
6	8	6872.875	
7	10	7286.29	
8	15	7332.75	
9	20	7565.63	
10	25	7645.413	
11	30	7725.45	
12	40	7740.328	
13	50	7684.567	
14	60	7705.628	
15	80	7718.365	
16			

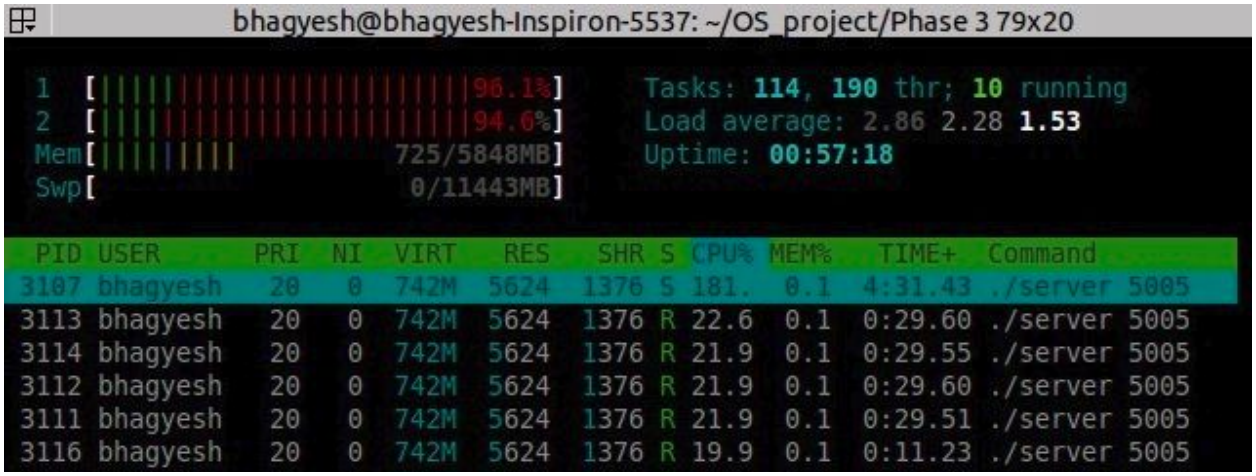
Average response time vs Load:



Values noted during the experiment on which the above graph is plotted:

	A	B	C
1	Load	Avg Response Time (in microsec)	
2	0	0	
3	1	241.749	
4	3	451.766	
5	5	507.996	
6	8	505.571	
7	10	529.887	
8	15	538.742	
9	20	556.321	
10	25	587.358	
11	30	760.824	
12	40	1609.728	
13	50	1705.59	
14	60	1836.582	
15	80	2019.172	
16			

Snapshot when the system hits the saturation during the Book requests:



The above graphs shows as number of users increases, average throughput flattens out and response time starts increasing. The utilization of the resources increases when the number of users increases, which can be viewed using system monitor and htop.

The system hits saturation when the number of users are 25 for the Book requests. During Book requests the bottleneck for the system is the CPU consumption by the front end server code. The capacity of the system at this time is around 7700 req/sec.

Comparing it with Phase 2 (Mysql) (for Book requests only):

In case of mysql, the system hits the saturation when number of users were 300 for Book requests and the capacity of the system at this time was around 420 req/sec.

The performance is increased drastically as the redis inserts the data of booking requests of client in RAM which takes less time compared to insertion in disk. RAM is fast for I/O and disks are comparatively slow. So we see a huge increase in throughput.

However the system now hits the saturation when users are only 25 because:-
In case of mysql, mysql takes time in disk I/O (for thread's request) and during this time CPU is not consumed by that thread (However during this time it is consumed by other threads). But in case of redis the I/O takes very less time as it fulfill request in RAM, so CPU is consumed almost every time. So CPU consumption in case of redis is very much higher, therefore system hits the saturation early.

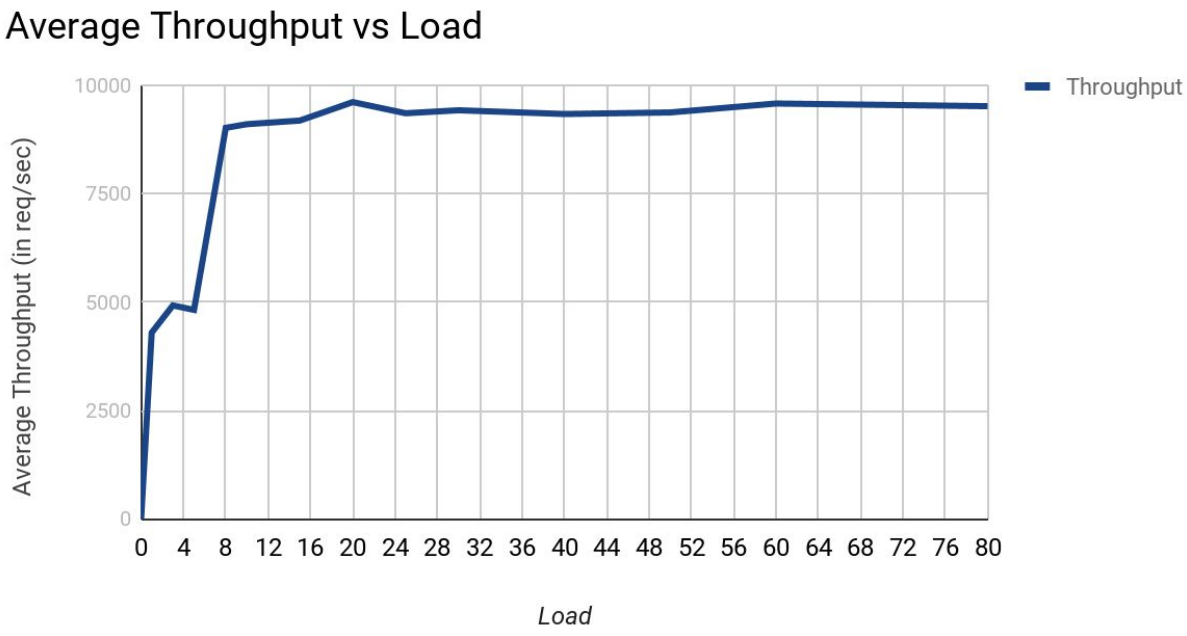
In case of mysql the saturation occurs for high number of users as at this time even if CPU is not consumed by some thread (because of I/O request), there are many more threads which are available for CPU consumption which will eventually saturate the system.

The response time is decreased drastically in case of redis as the number of requests being fulfilled are increased by a large number as insertion is done in RAM which takes very less time.

In case of mysql response time was hundreds of milliseconds while in redis response time is hundreds of microseconds.

2) When load is mixture of Book (70%) and Cancel Requests (30%):

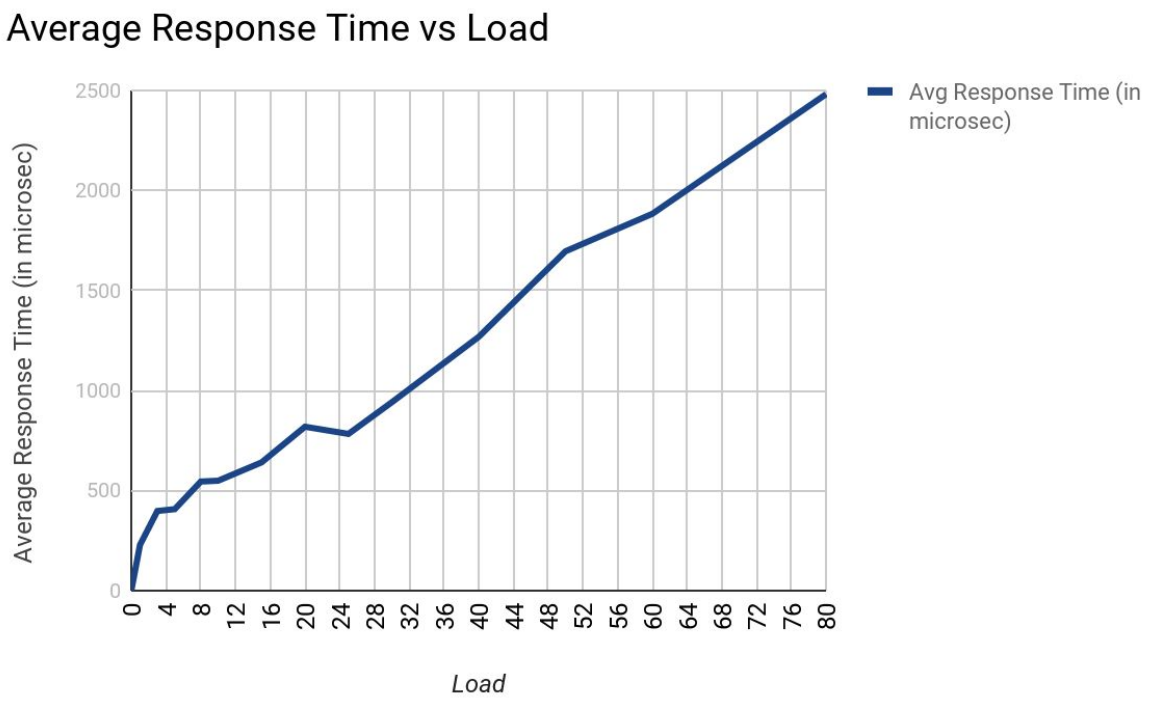
Average throughput vs Load:



Values noted during the experiment on which the above graph is plotted:

	A	B	C
1	Load	Throughput	
2	0	0	
3	1	4297.925	
4	3	4925.475	
5	5	4823.087	
6	8	9022.037	
7	10	9106.216	
8	15	9189.487	
9	20	9615.35	
10	25	9358.491	
11	30	9427.05	
12	40	9341.12	
13	50	9376.925	
14	60	9582.627	
15	80	9521.163	
16			

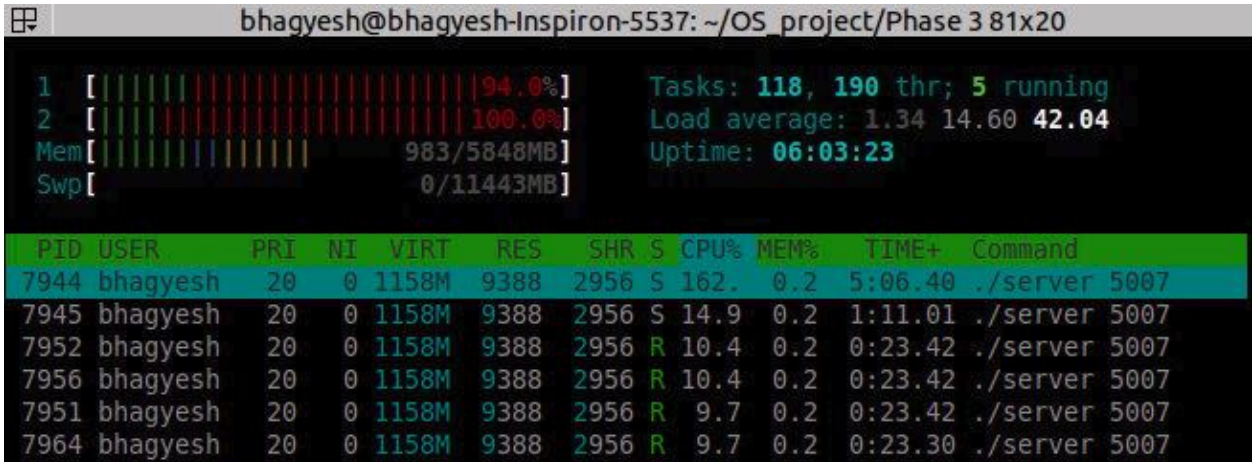
Average response time vs Load:



Values noted during the experiment on which the above graph is plotted:

	A	B	C
1	Load	Avg Response Time (in microsec)	
2	0	0	
3	1	228.81	
4	3	397.985	
5	5	407.216	
6	8	545.152	
7	10	549.381	
8	15	640.915	
9	20	819.121	
10	25	783.152	
11	30	941.516	
12	40	1268.183	
13	50	1696.026	
14	60	1883.272	
15	80	2481.096	
16			

Snapshot when the system hits the saturation during the mixture of Book and Cancel requests:



The above graphs shows as number of users increases, average throughput flattens out and response time starts increasing. The utilization of the resources increases when the number of users increases.

The system hits saturation when the number of users are 25 for the mixture of Book and Cancel requests. During this the bottleneck for the system is the CPU consumption by the front end server code. The capacity of the system at this time is around 9450 req/sec.

Comparing it with Phase 2 (Mysql) (for mixture of Book and Cancel Requests):

In case of mysql, the system hits the saturation when number of users were 300 for mixture of Book and Cancel requests and the capacity of the system at this time was around 415 req/sec.

The performance is increased drastically as the redis inserts the data of booking requests of client in RAM which takes less time compared to insertion in disk. Similarly for cancel, in case of mysql the searching from disk takes long time as compared to redis which just search RAM and deletes the entry. RAM is fast for I/O and disks are comparatively slow. So we see a huge increase in throughput.

However the system now hits the saturation when users are only 25 because:-
In case of mysql, mysql takes time in disk I/O (for thread's request) and during this time CPU is not consumed by that thread (However during this time it is consumed by other threads). But in case of redis the I/O takes very less time as it fulfill request in RAM, so CPU is consumed almost every time. So CPU consumption in case of redis is very much higher, therefore system hits the saturation early.

In case of mysql the saturation occurs for high number of users as at this time even if CPU is not consumed by some thread (because of I/O request), there are many more threads which are available for CPU consumption which will eventually saturate the system.

The response time is decreased drastically in case of redis as the number of requests being fulfilled are increased by a large number as insertion is done in RAM which takes very less time. And also cancel request is completed by deleting corresponding entry in RAM which again takes less time.

In case of mysql response time was hundreds of milliseconds while in redis response time is hundreds of microseconds.