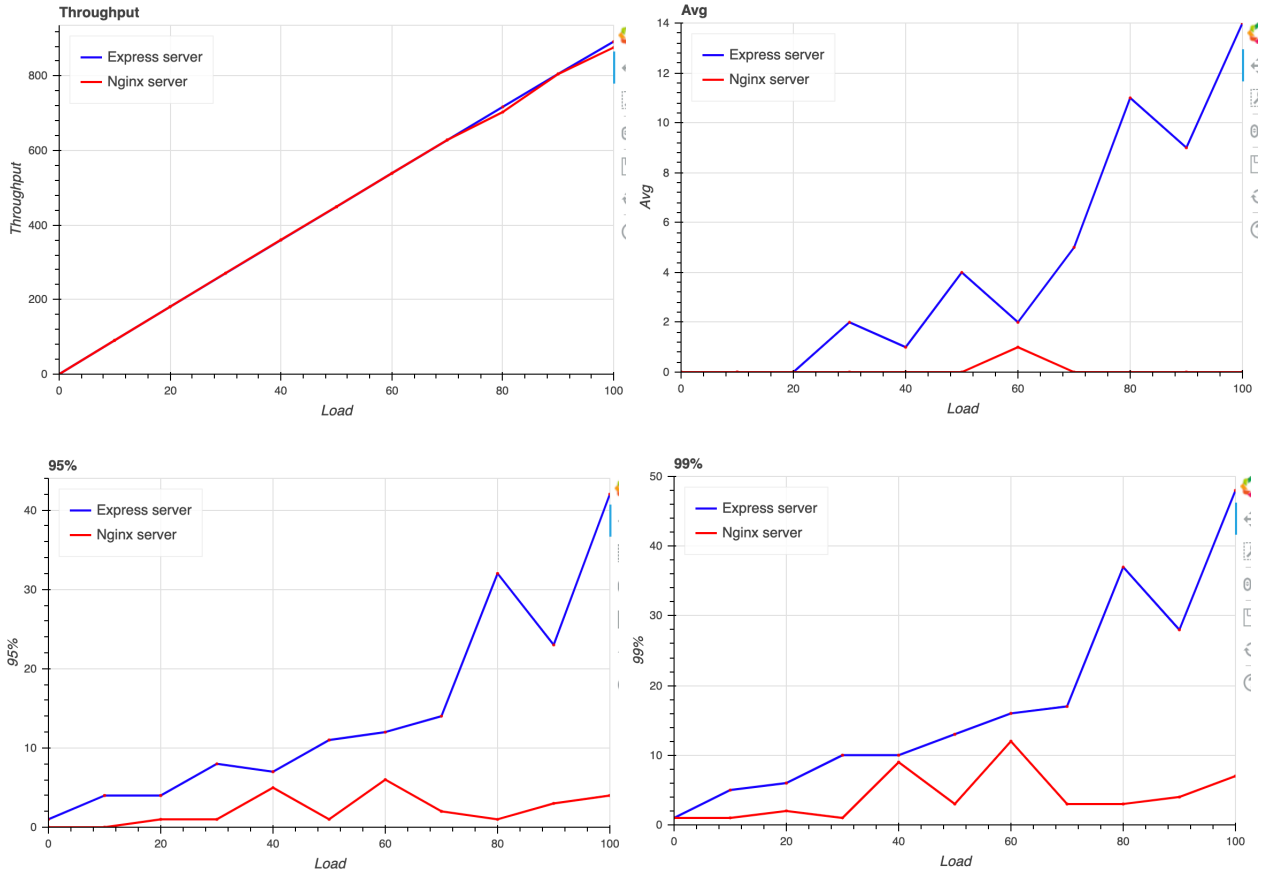
Handson3



Question 2: Please try the command by yourself and draw some curves to show the trend of RT and throughput under different loads.

测试结果如上图蓝线所示

Question 3: Please describe and explain the phenomenon you find in the curve you have just drawn.

随着负载增加，吞吐量线性增加，说明在0~100个用户的负载下该系统没有到达吞吐量的瓶颈，吞吐量可以随着负载增长而线性增长。同时，随着负载增加，在0~20个用户的负载下系统的响应时间基本不变，在20个用户以上负载下系统的响应时间线性增加。这说明在0~20个用户时，每一个请求被直接处理，所以响应时间基本不变；在20个用户以上的负载时，有一部分请求无法被立刻处理，并被放入一个队列中等待，所以请求的响应时间延长；随着并发数增加，需要进入队列的请求越多，等待队列越长，请求的平均响应时间越长。

Question 4: Try to optimize the static page accessing response time using `Nginx`. Please briefly describe what you have done and draw the curves to show the effect of your optimization.

在sockshop同一个bridge网络中创建了一个Nginx容器，Nginx将静态资源请求和动态服务请求分离，分别直接将资源发送给用户和转发动态服务请求到frontend容器负责转发到其他后端服务容器。具体配置如下：

```
server {
        listen 80;
        server_name localhost;
        location ~* \.
(html|css|map|eot|svg|ttf|woff|otf|png|woff2|gif|jpg|xcf|jpeg|js)$ {
                root /public/;
        }
        location / {
                proxy_pass http://frontend:8079;
        }
}
```

Nginx进程监听80端口，将每一个Http请求进行解析，若后缀符合上述正则表达式，则在本地/public文件夹下寻找对应静态资源并返回给用户；若后缀不符合上述正则表达式，则认为该请求是动态服务请求，转发给frontend容器，即转发给后端路由的容器。创建该容器的指令如下：

```
sudo docker run --rm --network handson3 --name nginx-router -p 8080:80 -v
`pwd`/public:/public -v `pwd`/config:/etc/nginx/conf.d -v
`pwd`/logs:/var/log/nginx --add-host=frontend:172.26.0.12 -d nginx
```

创建Nginx容器时，指定网络为sockshop同一个网络，将主机的8080端口映射到该容器Nginx进程的80监听端口，将静态资源目录public、Nginx配置文件、Nginx日志目录挂载到容器的对应位置，将该容器中frontend域名解析到实际frontend容器的ip地址172.26.0.12。

测试结果如上图所示，其中蓝线为load-test对主机的30000端口，即frontend容器的8079端口（express server）进行测试；红线为load-test对主机的8080端口，即Nginx容器的80端口（Nginx server）进行测试。

对上图分析可知，在0~100的用户负载下，Nginx处理静态资源请求的吞吐量和express相同。在0~100的用户负载下，Nginx处理静态资源请求的响应时间基本保持不变，并且明显小于express处理请求的响应时间，说明了Nginx对单个静态资源请求的处理速度要比express更加快，在0~100的用户负载下所有请求能够被直接处理，不需要进入队列。

Question 5: Try to optimize the performance of the application under this workload with load-balancing. Please briefly describe your modification. You also need to re-run `load-test.sh` and generate figures to show how well your optimization works.

添加redis容器作为frontend的session-db

```
sudo docker run --rm --name session-db --network handson3 --ip=172.26.0.12 -d
redis
```

启动两个frontend容器，并将他们的环境变量SESSION_REDIS设为true

```
sudo docker run --rm --name frontend --network handson3 --ip=172.26.0.13 --
add-host=catalogue:172.26.0.3 --add-host=orders:172.26.0.5 --add-
host=carts:172.26.0.7 --add-host=user:172.26.0.9 --add-
host=shipping:172.26.0.10 --add-host=payment:172.26.0.11 --add-host=session-
db:172.26.0.12 -e SESSION_REDIS=true -d dplsming/sockshop-frontend:0.1
```

```
sudo docker run --rm --name frontend-1 --network handson3 --ip=172.26.0.14 --
add-host=catalogue:172.26.0.3 --add-host=orders:172.26.0.5 --add-
host=carts:172.26.0.7 --add-host=user:172.26.0.9 --add-
host=shipping:172.26.0.10 --add-host=payment:172.26.0.11 --add-host=session-
db:172.26.0.12 -e SESSION_REDIS=true -d dplsming/sockshop-frontend:0.1
```

启动两个nginx容器，其中nginx-router-nolb的配置与Question 4中的配置相同

```
sudo docker run --rm --network handson3 --ip=172.26.0.18 --name nginx-router-
nolb -p 8080:80 -v `pwd`/public:/public -v `pwd`/config-nolb:/etc/nginx/conf.d
-v `pwd`/logs:/var/log/nginx --add-host=frontend:172.26.0.13 -d nginx
```

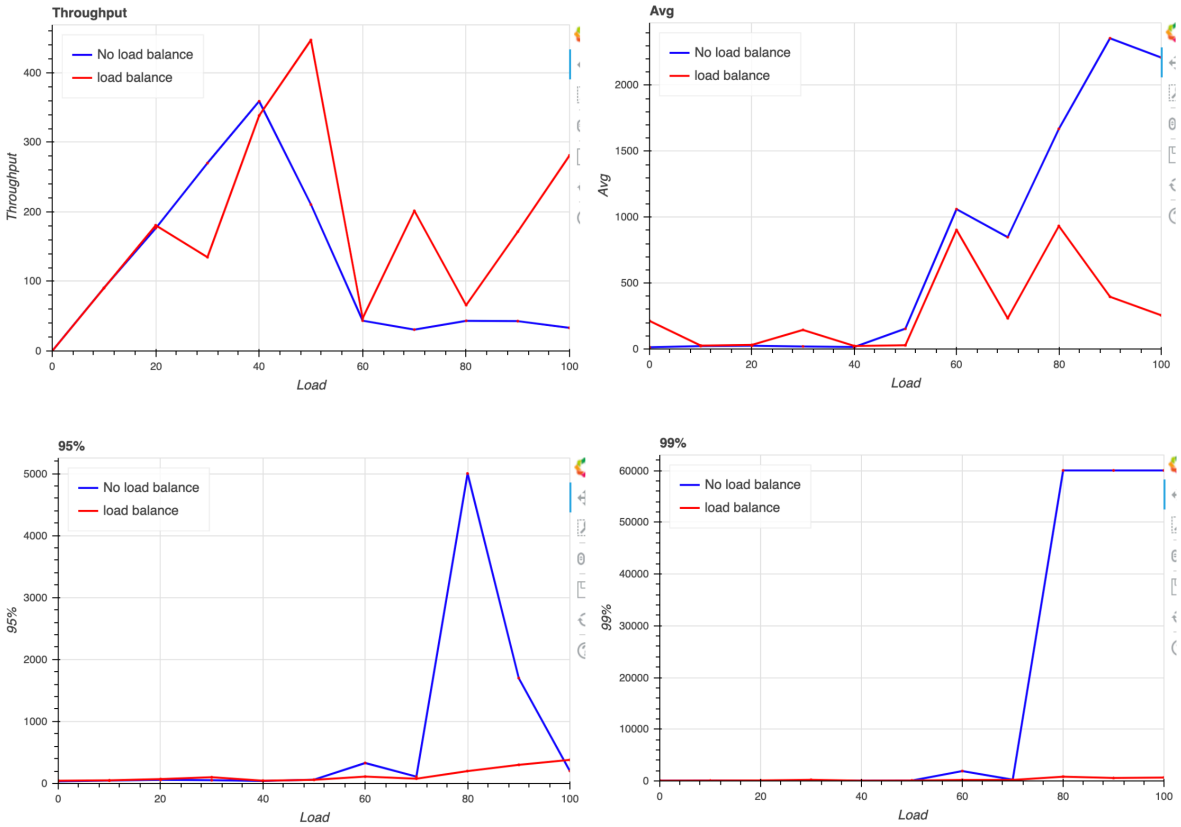
```
sudo docker run --rm --network handson3 --ip=172.26.0.19 --name nginx-router-
lb -p 8081:80 -v `pwd`/public:/public -v `pwd`/config-lb:/etc/nginx/conf.d -v
`pwd`/logs:/var/log/nginx --add-host=frontend:172.26.0.13 --add-host=frontend-
1:172.26.0.14 --add-host=frontend-2:172.26.0.15 --add-host=frontend-
3:172.26.0.16 --add-host=frontend-4:172.26.0.17 -d nginx
```

nginx-router-lb实现将请求在两个frontend容器之间负载均衡，配置如下。负载均衡策略为轮询。

```
upstream loadbalance {
        server frontend:8079;
        server frontend-1:8079;
        server frontend-2:8079;
        server frontend-3:8079;
        server frontend-4:8079;
}
server {
        listen 80;
        server_name localhost;
        location ~* \.
(html|css|map|eot|svg|ttf|woff|otf|png|woff2|gif|jpg|xcf|jpeg|js)$ {
                root /public/;
        }
        location / {
```

```
                proxy_pass http://loadbalance;
        }
}
```

测试结果如下图所示

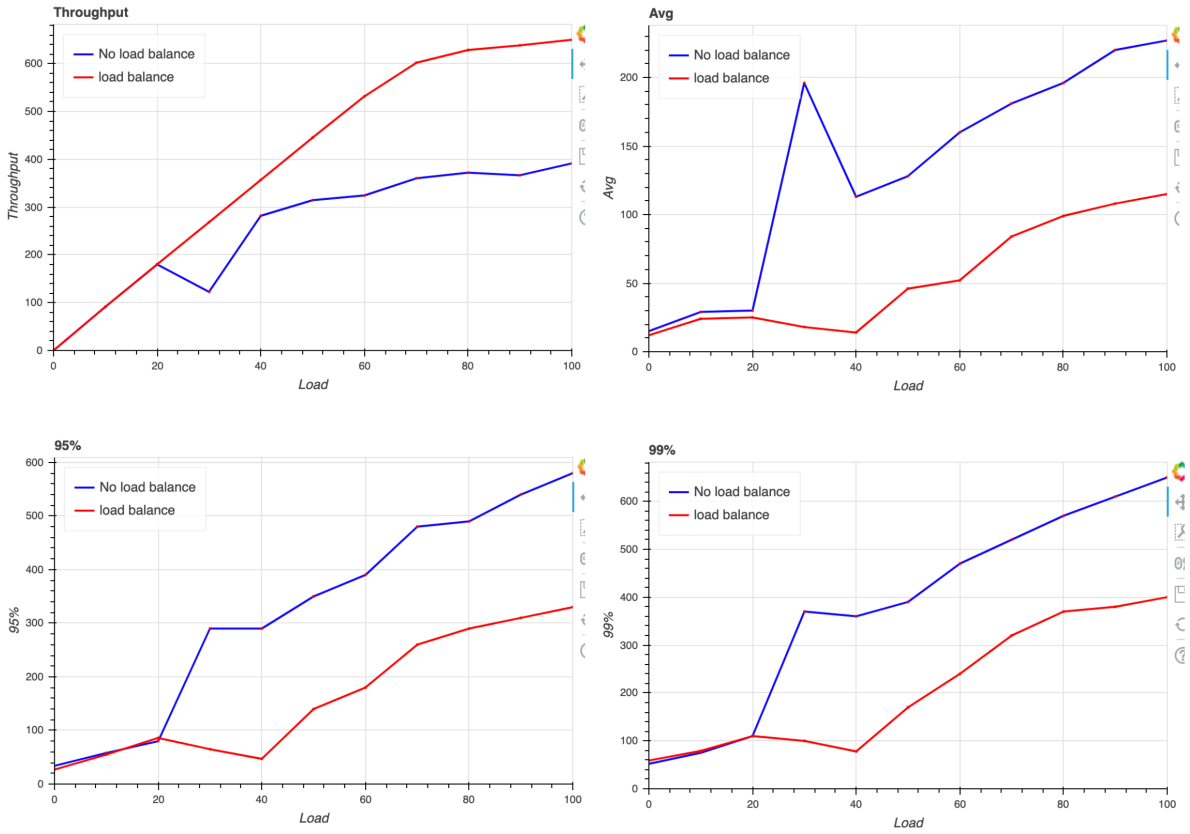


对上图分析可知，在没有负载均衡的情况下，40个用户负载就会导致系统的吞吐量到达瓶颈；当用户负载高于40个时，系统响应时间急剧升高；用户负载高于80个时，出现尾时延为60000ms的情况。这说明吞吐量到达瓶颈后，有一部分并发请求将无法及时进入队列，因此无法保证系统有稳定的响应时间；有一部分请求在60s后仍然没有被调度到，导致响应时间超过了nginx默认设置的60s超时时间，因此出现了尾时延为60000ms的情况。

在添加了frontend的负载均衡后，系统的吞吐量瓶颈仍然存在，但是比无负载均衡的系统的吞吐量有略微的提升；当用户负载高于50个时，系统响应时间开始升高，但是相比无负载均衡的系统响应时间有所降低；没有出现尾时延到达60000ms的情况。由此可以判断负载均衡的优化产生了一定的效果，缓解了系统到达吞吐量瓶颈后响应时间出现异常的情况。

Question 6: Try your optimization under the situation where all frontend containers are enforced with a CPU limitation using `--cpus 0.5` when created. Is your optimization more effective or less effective? Why?

优化效果比Question 5中更好。此时负载均衡和未负载均衡的测试结果如下图所示



通过上图和Question 5中的测试结果对比发现，在限定了frontend使用的CPU资源上限为0.5个CPU时，在未负载均衡的情况下，系统在40个用户负载以上到达吞吐量瓶颈后，吞吐量保持稳定，响应时间线性上升而不是急剧上升，没有出现60s响应时间的现象。对比上图和Question 5中对应用户负载下的吞吐量值和响应时间值，可以发现吞吐量和响应时间都有明显改善。结合对各个容器使用CPU资源的观察和分析，推断在不限定frontend的CPU资源时，frontend和后端服务容器、数据库容器的CPU资源发生了竞争，由此出现了上下文切换的开销，同时后端服务容器和数据库容器的CPU资源可能被frontend抢占，导致无法在稳定的时间内处理请求，产生了吞吐量骤降和响应时间骤升的情况。在限定了frontend可以占用的CPU资源之后，frontend与后端服务容器、后端数据库容器的CPU资源的竞争减少，后端服务容器和后端数据库容器可以稳定的处理请求。因此通过负载均衡解决frontend存在的瓶颈的效果会更加显著。

对上图进行分析，比较负载均衡的效果。可以发现吞吐量在40个用户负载以上仍然线性增长，到达80个用户负载出现瓶颈，有将近一倍的提升。响应时间在40个用户负载下下降了90%，在100个用户负载下下降了50%。这说明负载均衡对系统性能有了显著的提升。