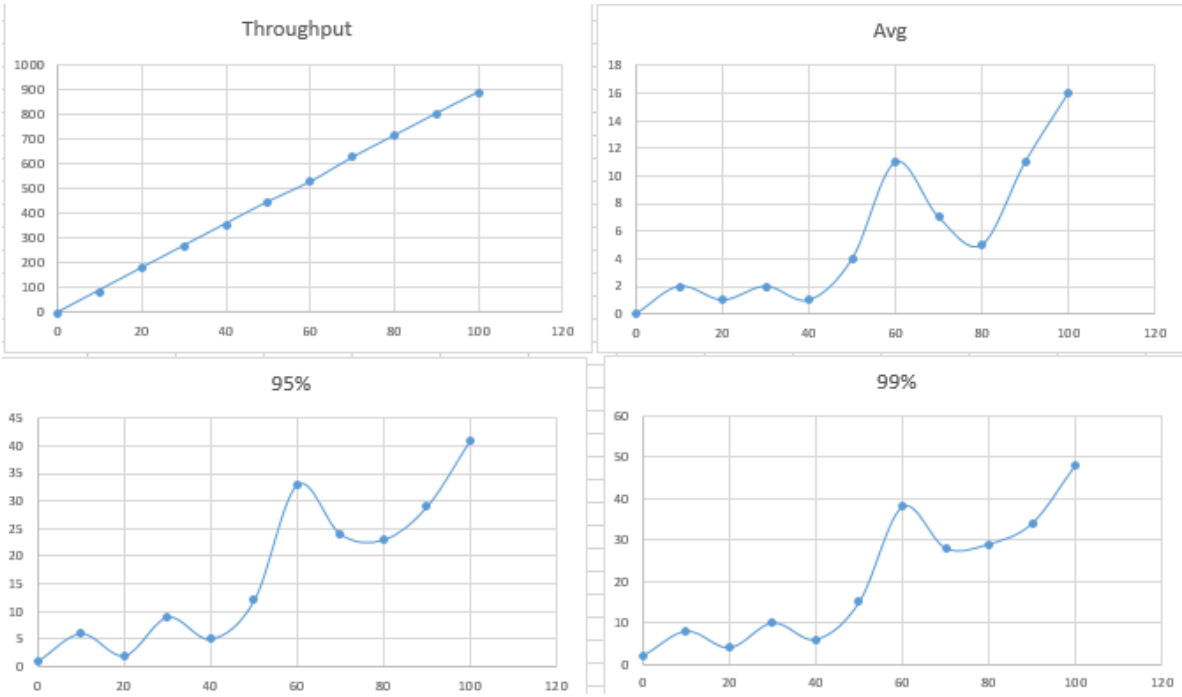
Handson3

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



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

现象：随着load（用户的增加），平均响应时间和95%、99%的响应时间整体呈上升趋势，从40个用户增加到60个用户的时候增长明显。吞吐量随load的增加而线性增加。

原因：吞吐量随load线性增加是因为性能还没达到服务器的性能瓶颈，用户越多吞吐量越大；同时，因为服务器处理一个请求的时间基本是固定的，用户越多latency越大，返回时间也就越长

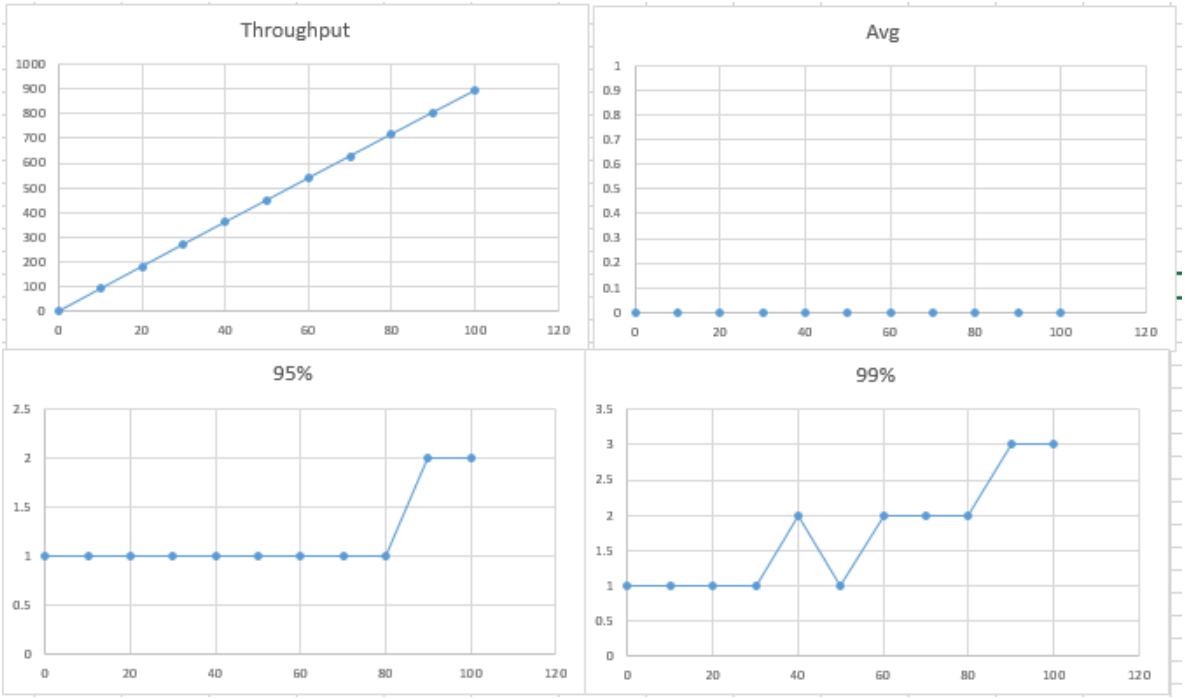
Question 4: Try to optimize the static page accessing response time using `Nginx`. Please briefly describe what you have done and draw the curves to show the effect of your optimization.

我用nginx配置了动静态资源分离，配置文件如下：

```
server {
    listen 8090;
    root /home/ubuntu/cse/handson3/frontend/public;
    index index.html index.htm index.nginx-debian.html;
    server_name ktyweavesock;
    # 正则匹配到则为静态请求
    location ^~ /css/ {

    }
    location ^~ /fonts/ {

    }
```

```
    location ^~ /img/ {

    }
    location ^~ /items/ {

    }
    location ^~ /js/ {

    }
    location ~ \.(html|png)$ {

    }
    # 如果上面的都不匹配，则为动态请求，转发给8080
    location / {
        proxy_pass http://localhost:8080;
    }
}
```

nginx监听8090端口，（即我们浏览器访问时应输入8090端口），然后对需要访问的资源进行正则匹配，如果以/css/、/fonts/、/img/、/items/、/js/开头或者以.html、.png结尾，则是静态资源（这是通过观察frontend/public中的文件得出的结论），直接返回`/home/ubuntu/cse/handson3/frontend/public`中对应的文件。否则为动态请求，转发给8080端口（docker容器起frontend镜像时使用的端口）。



可以看到优化后，平均响应时间降为0，最长的也是个位数级别的，效果还是很明显的。

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.

我多开了两个frontend容器，分别跑在8081端口和8082端口，然后在nginx中加上负载均衡，采用的是轮询策略：

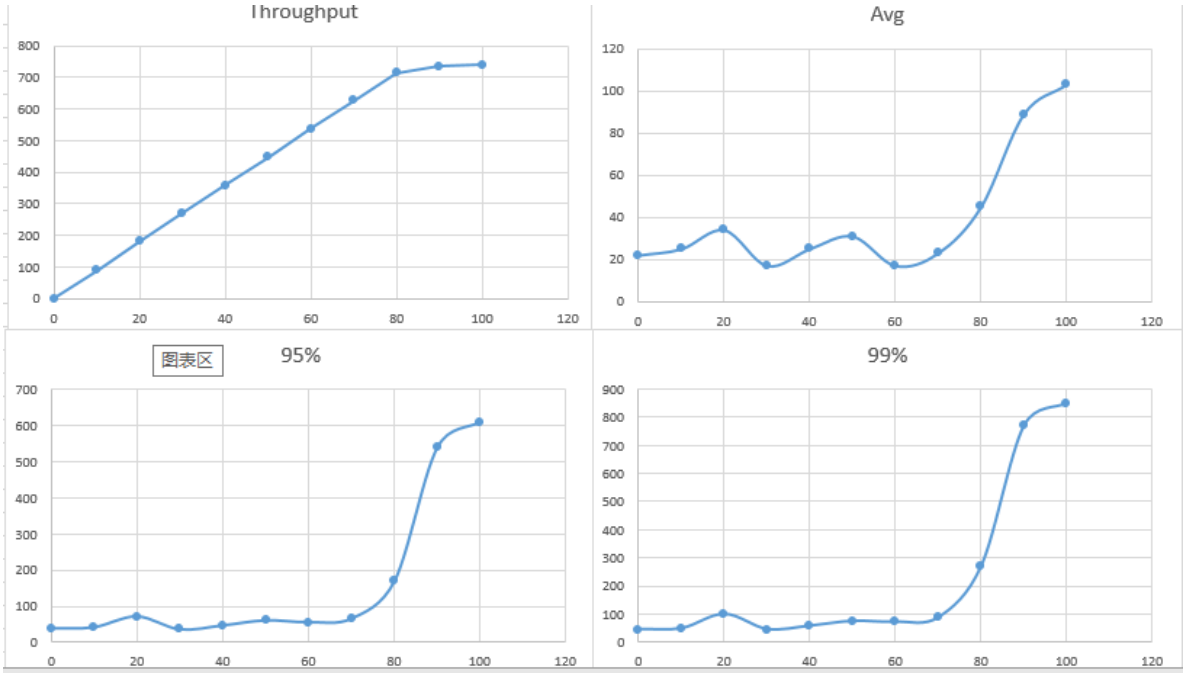
```
# 负载均衡动态服务器组
upstream dynamic_server {
    server 172.30.1.0:8079;
    server 172.30.1.1:8079;
    server 172.30.1.2:8079;
}
```

静态资源的请求不变，动态资源请求从固定地转发到8080改为转发到上面定义的`dynamic_server`：

```
location / {
        proxy_pass http://dynamic_server;
    }
```

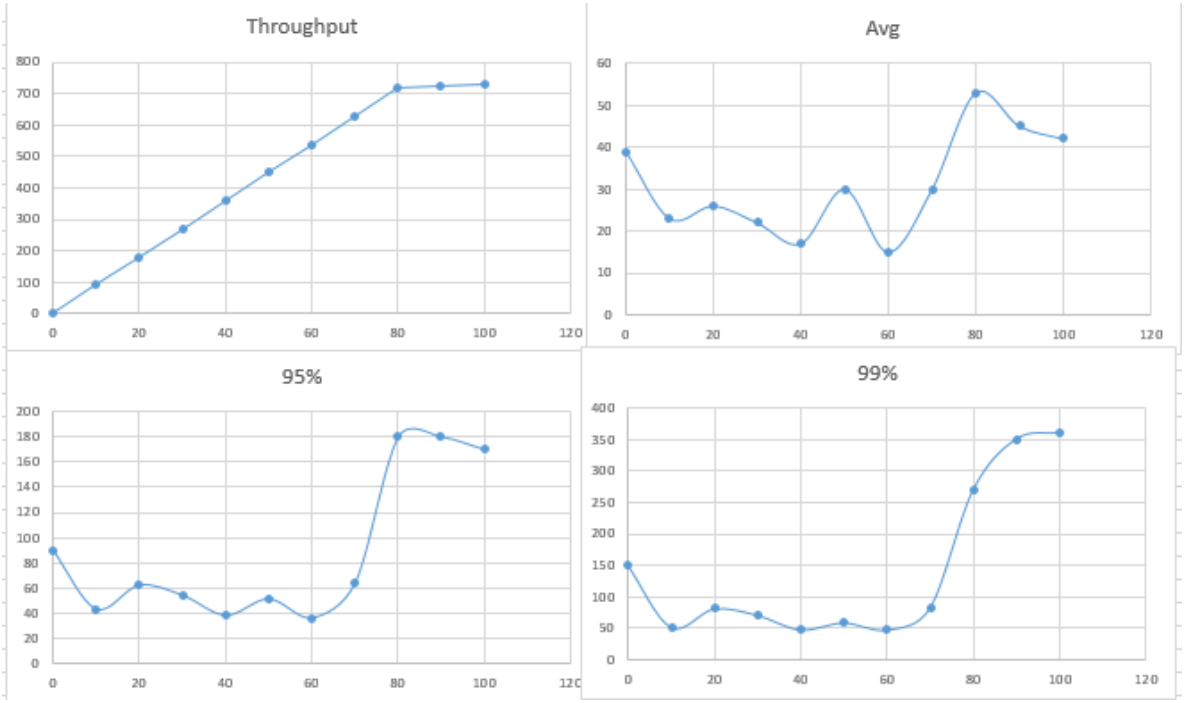
效果对比：

使用负载均衡之前：



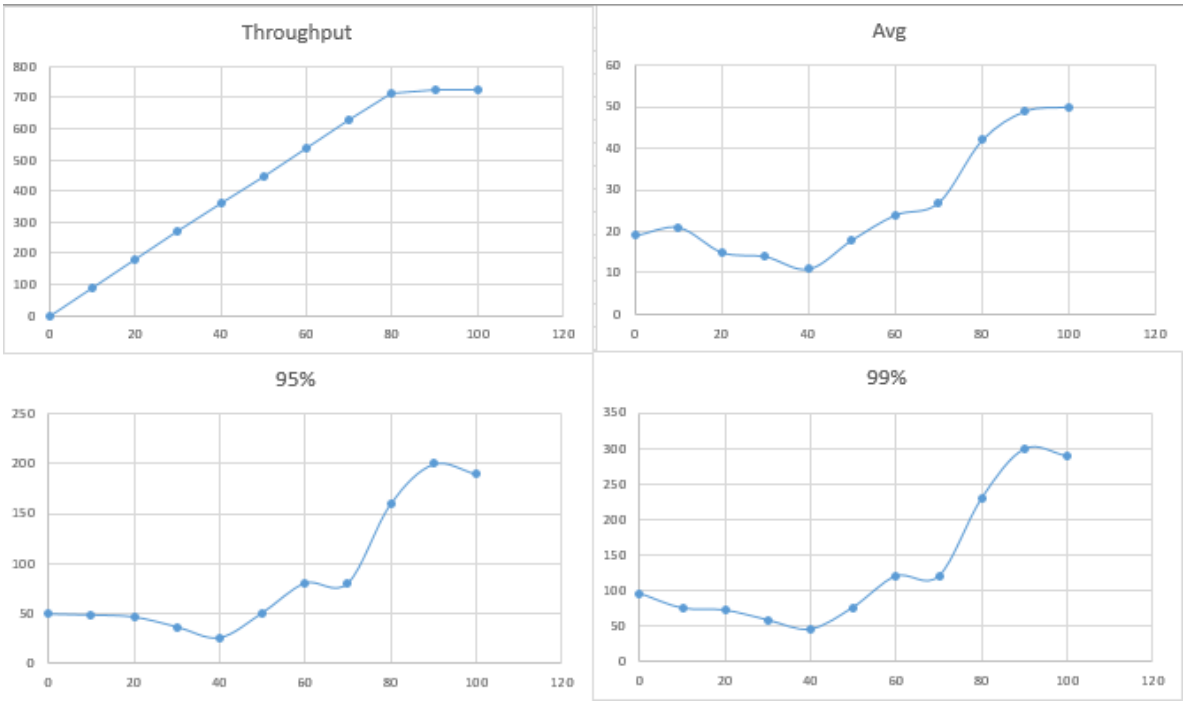
可以看到平均响应时间在20ms以上，当用户数达到80时，frontend性能达到瓶颈，吞吐量不再线性增长，响应时间急剧增加。

使用负载均衡以后：



可以看到曲线的大致形状基本和不用负载均衡一样（不知道为啥load为0的时候会有这么大的响应时间……）。throughput完全一样，但是latency在数值上与不用负载均衡相比性能有很大的提升，平均latency在10-60ms之间，load为100时，完成99%任务的latency也只有350ms，与不使用负载均衡的850ms，性能提升很大。

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?



限制cpus后，more effective了，曲线和数据基本一样，但是可以看到latency还是略微减少了。

原因：在case2的高并发请求下，frontend本身需要很多cpu资源来“分发”任务给后端，如果不限制frontend的cpu资源使用，那么frontend“分发”给后端的过程就占用了大量cpu，导致实际处理请求的后端进程分到的cpu资源就少了，导致整体请求的latency都有所增加，（有点frontend先分发任务给后端，全部派完之后端再处理的感觉）。而限制了frontend的cpu为0.5后，就变成frontend边分发给后端，后端边处理了，就变成流水线了，因此throughput不变，latency减少了。