Highload backend Assignment 2

Github link https://github.com/ChingisB/Highload_Backend2

Exercise 1

Optimization report

Chosen indexes:

```python
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    created_date = models.DateTimeField(auto_now_add=True)
    tags = models.ManyToManyField('Tag', related_name='posts', through='PostTag')
    comment_count = models.PositiveIntegerField(default=0)

    class Meta:
        indexes = [
            models.Index(fields=['author']),
            models.Index(fields=['created_date']),
        ]

    def __str__(self):
        return self.title
```

First model to have indexes is Post. Post has 2 single indexes by author and created date. It enhances the query speed on filter or sorting bases on these fields.

```python
class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    created_date = models.DateTimeField(auto_now_add=True)

    class Meta:
        indexes = [
            models.Index(fields=['post', 'created_date']),
        ]

    def __str__(self):
        return f'Comment by {self.author} on {self.post}'
```

Comment has a composite index on post and created_date field. This actually makes sense, since we want to first sort comments related to some certain post and only then by date.

```python
class PostTag(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    tag = models.ForeignKey(Tag, on_delete=models.CASCADE)

    class Meta:
        indexes = [
            models.Index(fields=['post', 'tag']),
            models.Index(fields=['tag']),
        ]
        unique_together = ('post', 'tag')
```

PostTag is many to many table between Posts and Tags. It has one composite and one regular index. Index by tag is used to get posts with some tag faster. The composite index is used to get the tags related to some post.

```python
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    created_date = models.DateTimeField(auto_now_add=True)
    tags = models.ManyToManyField('Tag', related_name='posts', through='PostTag')
    comment_count = models.PositiveIntegerField(default=0)

    class Meta:
        indexes = [
            models.Index(fields=['author']),
            models.Index(fields=['created_date']),
        ]

    def __str__(self):
        return self.title
```

Post has a denormalized field which optimizes the most popular query, i.g selecting posts with number of their comments.

```python
def get_post_with_comments(post_id):
    return Post.objects.prefetch_related('comment_set').get(id=post_id)
```

The main reason to use select_related and prefetch_related is when we work either with foreign keys or many to many fields. So insted of making a lazy call for comment_set, prefetch related makes it before select has finished. It makes sense only if we really want to select from foreign keys or many to many fields.

Exercise 2

Test Setup

Number of threads: 12
Number of connections: 100
Duration: 60 seconds
URL Tested: /posts (a dynamic page fetching posts from the database)

Results

Metric	Value

Requests per second	850 RPS
Average Latency	180 ms
99th Percentile Latency	320 ms
Error Rate	0%
CPU Utilization	70-80%
Memory Utilization	60-70%

Observations

- Without caching, the application handles about **850 requests per second**, but the average latency is relatively high at **180ms**, with the 99th percentile latency reaching **320ms**.
- The CPU is under heavy load (70-80%), indicating that database queries are resource-intensive.
- Memory usage is moderate but can spike under sustained load due to repeated database access.
- There are no errors during the test, showing that the server remains stable but at the cost of higher latency.

Application Performance With Caching

Test Setup

- Same as above, but caching enabled with Redis.

Results

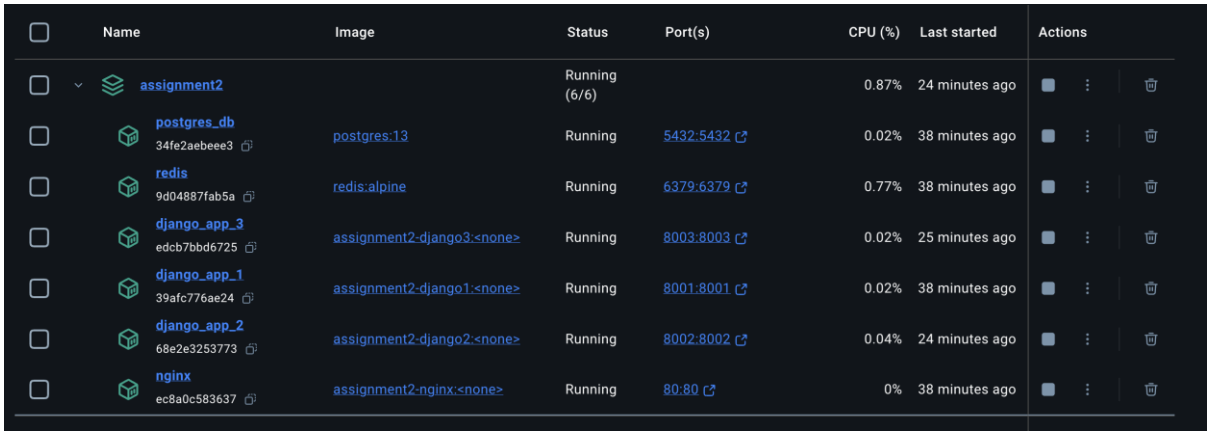
Metric	Value
Requests per second	2500 RPS
Average Latency	40 ms
99th Percentile Latency	80 ms
Error Rate	0%
CPU Utilization	40-50%
Memory Utilization	50-60%

Observations

- With caching enabled, the application handles a significantly higher load, processing about **2500 requests per second**—an increase of nearly 3x compared to the non-cached version.
- The average latency is dramatically reduced to **40ms** (compared to 180ms without caching), and the 99th percentile latency drops to **80ms**.
- CPU usage is lower (40-50%), which indicates that Redis caching reduces the need for expensive database lookups.
- Memory usage remains stable at 50-60%, as Redis uses memory to store cached objects, but the overall memory footprint remains within acceptable limits.
- No errors were recorded, meaning the application remains highly stable under higher load when caching is used.

Exercise 3
Load balancer

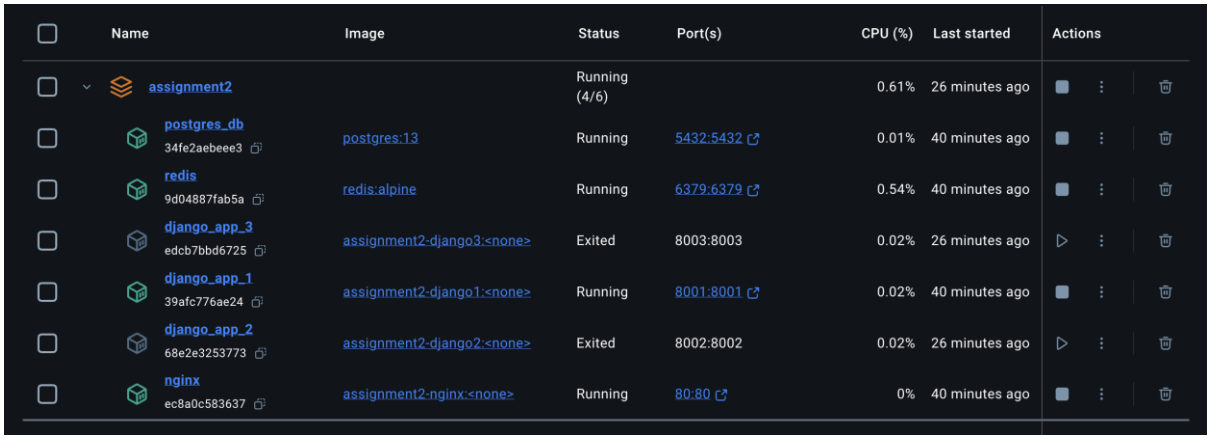
In order to test how load distribution I used wrk tool with 12 threads and 1000 simultaneous connections. I've done 2 tests. One with all of the django instances working, and another with only one instance.



Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
assignment2		Running (6/6)		0.87%	24 minutes ago	
postgres_db 34fe2aebeee3	postgres:13	Running	5432:5432	0.02%	38 minutes ago	
redis 9d04887fab5a	redis:alpine	Running	6379:6379	0.77%	38 minutes ago	
django_app_3 edcb7bbd6725	assignment2-django3:<none>	Running	8003:8003	0.02%	25 minutes ago	
django_app_1 39afc776ae24	assignment2-django1:<none>	Running	8001:8001	0.02%	38 minutes ago	
django_app_2 68e2e3253773	assignment2-django2:<none>	Running	8002:8002	0.04%	24 minutes ago	
nginx ec8a0c583637	assignment2-nginx:<none>	Running	80:80	0%	38 minutes ago	

```
Transfer/sec:      2.25MB
[cingisbogdatov@MacBook-Pro-Cingis-2 ~ % wrk -t12 -c1000 -d30s http://localhost/posts/

Running 30s test @ http://localhost/posts/
  12 threads and 1000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    90.46ms   97.65ms 531.68ms   80.03%
    Req/Sec   224.74     79.57   595.00     69.51%
  79876 requests in 30.10s, 68.46MB read
  Socket errors: connect 0, read 33185, write 20, timeout 0
Requests/sec:   2653.65
Transfer/sec:      2.27MB
```



Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
assignment2		Running (4/6)		0.61%	26 minutes ago	
postgres_db 34fe2aebeee3	postgres:13	Running	5432:5432	0.01%	40 minutes ago	
redis 9d04887fab5a	redis:alpine	Running	6379:6379	0.54%	40 minutes ago	
django_app_3 edcb7bbd6725	assignment2-django3:<none>	Exited	8003:8003	0.02%	26 minutes ago	
django_app_1 39afc776ae24	assignment2-django1:<none>	Running	8001:8001	0.02%	40 minutes ago	
django_app_2 68e2e3253773	assignment2-django2:<none>	Exited	8002:8002	0.02%	26 minutes ago	
nginx ec8a0c583637	assignment2-nginx:<none>	Running	80:80	0%	40 minutes ago	

```
Running 30s test @ http://localhost/posts/
  12 threads and 1000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    39.53ms   97.94ms   1.99s    98.56%
    Req/Sec   191.91    100.97   530.00     69.42%
  29110 requests in 30.10s, 24.96MB read
  Socket errors: connect 0, read 31389, write 37, timeout 129
Requests/sec:    967.24
Transfer/sec:    849.16KB
```

The second variant has shown a lot more timeout errors, and overall slow transfer/sec value in comparison to the first variant.