Highload - Assignment 2

Exercise 1: Database Design and Optimization

Schema Design

Using “models” methods for create field with specific types and attributes

and set associated tables like, manyToMany, ForeignKey

Indexing

Set indexes with method Index for increase search perfomance

```python
class User(models.Model):
    username = models.CharField(max_length=150, unique=True)
    email = models.EmailField(unique=True)
    password = models.CharField(max_length=128)
    bio = models.TextField(blank=True, null=True)

    def set_password(self, raw_password):
        self.password = make_password(raw_password)

    def check_password(self, raw_password):
        from django.contrib.auth.hashers import check_password
        return check_password(raw_password, self.password)

    def __str__(self):
        return self.username

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', blank=True)

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

```python
class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')
    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']),
        ]

class Tag(models.Model):
    name = models.CharField(max_length=50, unique=True)

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

Query Optimization

```python
def get_all(request):
    posts_with_comments = Post.objects.select_related('author').prefetch_related(
        Prefetch('comments', queryset=Comment.objects.select_related('author'))
    )
```

```sql
SELECT
  "blog_post"."id",
  "blog_post"."title",
  "blog_post"."content",
  "blog_post"."author_id",
  "blog_post"."created_date",
  "blog_user"."id",
  "blog_user"."username",
  "blog_user"."email",
  "blog_user"."password",
  "blog_user"."bio"
FROM
  "blog_post"
  INNER JOIN "blog_user" ON (
    "blog_post"."author_id" = "blog_user"."id"
  );
args =();
alias = default (0.000)
SELECT
  "blog_comment"."id",
  "blog_comment"."post_id",
  "blog_comment"."author_id",
  "blog_comment"."content",
  "blog_comment"."created_date",
  "blog_user"."id",
  "blog_user"."username",
  "blog_user"."email",
  "blog_user"."password",
  "blog_user"."bio"
FROM
  "blog_comment"
  INNER JOIN "blog_user" ON (
    "blog_comment"."author_id" = "blog_user"."id"
  )
WHERE
  "blog_comment"."post_id" IN (1, 2);
args =(1, 2);
alias = default
```

Method Post.objects.select_related(‘author’) add author data in one query to all posts

.prefetch_related(Prefetch(‘comments’, queryset = ...)) load first comments table and then exec query set

Comment.objects.select_related(‘author’) add author data to comments in one query to all comments

Indexes for author and post help to faster associate tables with INNER JOIN

Using denormalization its add additional duplicate values, for example: add author username to comments and post table.

For filter use “only” to select only required fields, query will be faster and lightweight

Exercise 2: Caching Strategies

Basic Caching

Using decorator cache_page from django lib

```python
from django.views.decorators.cache imp

@cache_page(60)
```

Template Fragment Caching

Set cache settings in html template,
user will get saved version of this html part

```html
<p>{{ post.content }}</p>
<p><strong>Автор:</strong> {{ post.author.username }}</p>
<p><strong>Дата создания:</strong> {{ post.created_date }}</p>

<h2>Последние комментарии</h2>

{% load cache %}
{% cache 600 'blog_post_'|add:post.id %}  <!-- 10 minutes -->

<ul>
    {% for comment in post.comments.all %}
    <li>
        <strong>{{ comment.author.username }}:</strong>
        {{ comment.content }} ({{ comment.created_date }})
    </li>
    {% empty %}
    <li>Нет комментариев.</li>
    {% endfor %}
</ul>

{% endcache %}
```

Low-Level Caching

Work with cache.get and cache.set

```python
def get_comment_count(post_id):
    cache_key = f'comment_count_{post_id}'
    comment_count = cache.get(cache_key)

    if comment_count is None:
        comment_count = Comment.objects.filter(post_id=post_id).count()
        cache.set(cache_key, comment_count, timeout=60)

    return comment_count
```

Cache Backend

Using outer database for caching - Redis

and set its in CONFIG

```
88] 04 Oct 15:52:03.832 # Warning: no config file specified, using t
 use D:\Progs\redis-latest\redis-server.exe /path/to/redis.conf

                        Redis 3.0.503 (00000000/0) 64 bi

                        Running in standalone mode
                        Port: 6379
                        PID: 30988

                            http://redis.io
```

```python
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
            'TIMEOUT': 60 * 5,
        }
    }
}
```

Performance Analysis

simulate requests with hey http tool
hey -n 1000 -c 100 <domain>

without view caching

```
D:\Progs\http-hey-tool>hey -n 1000 -c 100 http://localhost:8000/blog

Summary:
  Total:        11.4608 secs
  Slowest:      2.4022 secs
  Fastest:      0.1589 secs
  Average:      1.1298 secs
  Requests/sec: 87.2539

  Total data:   7999418 bytes
  Size/request: 8113 bytes

Response time histogram:
  0.159 [1]     |
  0.383 [8]     |■
  0.608 [15]    |■
  0.832 [46]    |■■■
  1.056 [207]   |■■■■■■■■■■■■■■■
  1.281 [548]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  1.505 [108]   |■■■■■■■■
  1.729 [27]    |■■
  1.954 [14]    |■
  2.178 [7]     |■
  2.402 [5]     |
```

with caching

```
D:\Progs\http-hey-tool>hey -n 1000 -c 100 http://localhost:8000/blog

Summary:
  Total:        7.3995 secs
  Slowest:      1.3836 secs
  Fastest:      0.1555 secs
  Average:      0.7177 secs
  Requests/sec: 135.1439

  Total data:   7958853 bytes
  Size/request: 8113 bytes

Response time histogram:
  0.155 [1]     |
  0.278 [2]     |
  0.401 [3]     |
  0.524 [24]    |■■■
  0.647 [381]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.770 [365]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.892 [102]   |■■■■■■■■■■■
  1.015 [34]    |■■■■
  1.138 [6]     |■
  1.261 [17]    |■■
  1.384 [46]    |■■■■■
```

Exercise 3: Load Balancing Techniques

Set Up a Basic Load Balancer

Upsteam create servers block for auto balancing between them

```nginx
upstream django_app {
    server localhost:8000;
    server localhost:8001;
}

server {
    listen 80;

    location / {
        proxy_pass http://django_app;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Session Management

ip_hash will be guarantee what requests from one user processing from one server

```nginx
upstream django_app {
    ip_hash;

    server localhost:8000;
    server localhost:8001;
}
```

Health Checks

For passive health checks, NGINX monitor requests and try to resume failed connections.

fail_timeout - max time for response
max_fails - max counts for fail conditions

```nginx
upstream django_app {
    ip_hash;

    server localhost:8000 slow_start=30s;
    server localhost:8001 max_fails=3 fail_timeout=30s;
}

server {
    listen 80;

    location / {
        proxy_pass http://django_app;

        health_check interval=10 fails=3 passes=2;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Perfomance

With load balancer and caching its faster in 4x times

```
D:\Progs\http-hey-tool>hey -n 1000 -c 100 http://localhost:1234/blog

Summary:
  Total:        3.4025 secs
  Slowest:      3.3994 secs
  Fastest:      0.0000 secs
  Average:      0.2387 secs
  Requests/sec: 293.8999

  Total data:   4707832 bytes
  Size/request: 4707 bytes

Response time histogram:
  0.000 [1]     |
  0.340 [847]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.680 [53]    |■■■
  1.020 [14]    |■
  1.360 [8]     |
  1.700 [23]    |■
  2.040 [17]    |■
  2.380 [5]     |
  2.719 [28]    |■
  3.059 [3]     |
  3.399 [1]     |
```