Backend for High Loaded Environment
Assignment 2

Prepared by: Merezhko Oleg
ID: 21B030698
Faculty: SITE
Major: Computer Systems and Software

Almaty, 2024

Exercise 1:

Task 3:

There are two ways to fetch all posts with their related comments: using `values` and `select_related` methods. First one will return dictionary-like objects with all necessary fields: Comment.objects.all().values('id', 'author', 'content', 'created_at', 'post__id', 'post__author', 'post__content', 'post__title')

Underlying query is: `SELECT … FROM "blog_comment" INNER JOIN "blog_post" ON ("blog_comment"."post_id" = "blog_post"."id")`

Second way will return normal objects with prefetched fields from related table: `Comment.objects.all().select_related('post').`

Underlying query is: `SELECT … FROM "blog_comment" INNER JOIN "blog_post" ON ("blog_comment"."post_id" = "blog_post"."id").`

So both ways use the same query, only difference is that second way give more usual for Django result.

Task 4:

Indexes can significantly improve query's performance. For example, index on model Post for filtering by author improved execution time by 8 times:

Without index – 0.480 ms.

```
>>> print(Post.objects.filter(author__username='osmer').all().explain(verbose=True, analyze=True))
Nested Loop  (cost=0.29..16.33 rows=4 width=470) (actual time=0.242..0.244 rows=1 loops=1)
  Output: blog_post.id, blog_post.author_id, blog_post.title, blog_post.content, blog_post.created_at
  ->  Index Scan using auth_user_username_6821ab7c_like on public.auth_user  (cost=0.14..8.16 rows=1 width=4) (actual time=0.235..0.235 rows=1 loops=1)
        Output: auth_user.id, auth_user.password, auth_user.last_login, auth_user.is_superuser, auth_user.username, auth_user.first_name, auth_user.last_name, auth_user.email, auth_use
r.is_staff, auth_user.is_active, auth_user.date_joined
        Index Cond: ((auth_user.username)::text = 'osmer'::text)
  ->  Index Scan using blog_post_author_id_dd7a8485 on public.blog_post  (cost=0.14..8.16 rows=1 width=470) (actual time=0.004..0.004 rows=1 loops=1)
        Output: blog_post.id, blog_post.title, blog_post.content, blog_post.created_at, blog_post.author_id
        Index Cond: (blog_post.author_id = auth_user.id)
Planning Time: 2.441 ms
Execution Time: 0.480 ms
```

With index – 0.057 ms.

```
>>> print(Post.objects.filter(author__username='osmer').all().explain(verbose=True, analyze=True))
Nested Loop  (cost=0.14..9.18 rows=1 width=470) (actual time=0.029..0.030 rows=1 loops=1)
  Output: blog_post.id, blog_post.author_id, blog_post.title, blog_post.content, blog_post.created_at
  Inner Unique: true
  Join Filter: (blog_post.author_id = auth_user.id)
  ->  Seq Scan on public.blog_post  (cost=0.00..1.01 rows=1 width=470) (actual time=0.011..0.012 rows=1 loops=1)
        Output: blog_post.id, blog_post.title, blog_post.content, blog_post.created_at, blog_post.author_id
  ->  Index Scan using auth_user_username_6821ab7c_like on public.auth_user  (cost=0.14..8.16 rows=1 width=4) (actual time=0.013..0.013 rows=1 loops=1)
        Output: auth_user.id, auth_user.password, auth_user.last_login, auth_user.is_superuser, auth_user.username, auth_user.first_name, auth_user.last_name, auth_user.email, auth_use
r.is_staff, auth_user.is_active, auth_user.date_joined
        Index Cond: ((auth_user.username)::text = 'osmer'::text)
Planning Time: 0.195 ms
Execution Time: 0.057 ms
```

Composite index on Comment table also have improved execution time.

Without index.

```
>>> print(Comment.objects.filter(post__id=1).all().explain(verbose=True, analyze=True))
Bitmap Heap Scan on public.blog_comment  (cost=4.19..12.66 rows=5 width=60) (actual time=0.013..0.014 rows=0 loops=1)
  Output: id, author_id, post_id, content, created_at
  Recheck Cond: (blog_comment.post_id = 1)
  ->  Bitmap Index Scan on blog_comment_post_id_580e96ef  (cost=0.00..4.19 rows=5 width=0) (actual time=0.012..0.012 rows=0 loops=1)
        Index Cond: (blog_comment.post_id = 1)
Planning Time: 0.104 ms
Execution Time: 0.041 ms
```

With index.

```
>>> print(Comment.objects.filter(post__id=1).all().explain(verbose=True, analyze=True))
Seq Scan on public.blog_comment  (cost=0.00..1.02 rows=1 width=60) (actual time=0.009..0.010 rows=0 loops=1)
  Output: id, author_id, post_id, content, created_at
  Filter: (blog_comment.post_id = 1)
  Rows Removed by Filter: 2
Planning Time: 0.077 ms
Execution Time: 0.022 ms
```

Denormalization can improve execution time of inner joins or repetitive queries with aggregate functions.

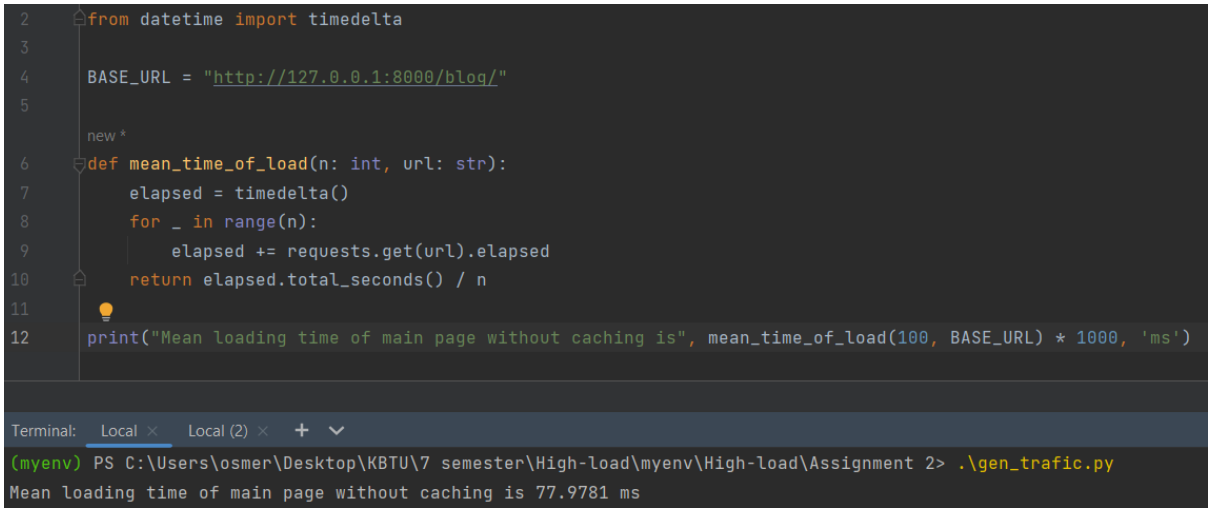
As shown above, `select_related` have 0 impact on single query as in both cases inner join is used. But for real applications `select_related` is beneficial as it lets use Django style iteration on elements and get data by object's fields. But there is a downside of such

approaches: they for each object load whole related objects. If we have, for example, 10 posts, each with 1 million comments, then each post will have 1 million repetitions in memory and connection with database will send a lot of repetitive data. In such case it is beneficial to retrieve in two separate queries posts and their comments and merge inside application.

Exercise 2:

Cache can significantly improve load time of rarely changing pages as main page. Here we can see almost 20 times speed up using Redis as cache backend.

Without cache mean time is 78 ms.



With cache: 4 ms.