

PJT 08

PJT 08 라이브

목차

- REST API – M:N
 - REST API 문서화
- Fixtures
- Improve query
 - 쿼리셋 이해하기
 - 필요하지 않은 것을 검색하지 않기
 - 한번에 모든 것을 검색하기

REST API – M:N

REST API 문서화

| drf-yasg 라이브러리

- “Yet another Swagger generator”
- API를 설계하고 문서화하는데 도움을 주는 라이브러리
- Swagger & OpenAPI 2.0 문서를 제공

| drf-yasg 라이브러리

- 설치 및 등록

```
$ pip install drf-yasg
```

```
# settings.py  
  
INSTALLED_APPS = [  
  
    ...  
    'drf_yasg',  
  
    ...  
]
```

drf-yasg 라이브러리

- swagger 관련 url 설정

```
# articles/urls.py

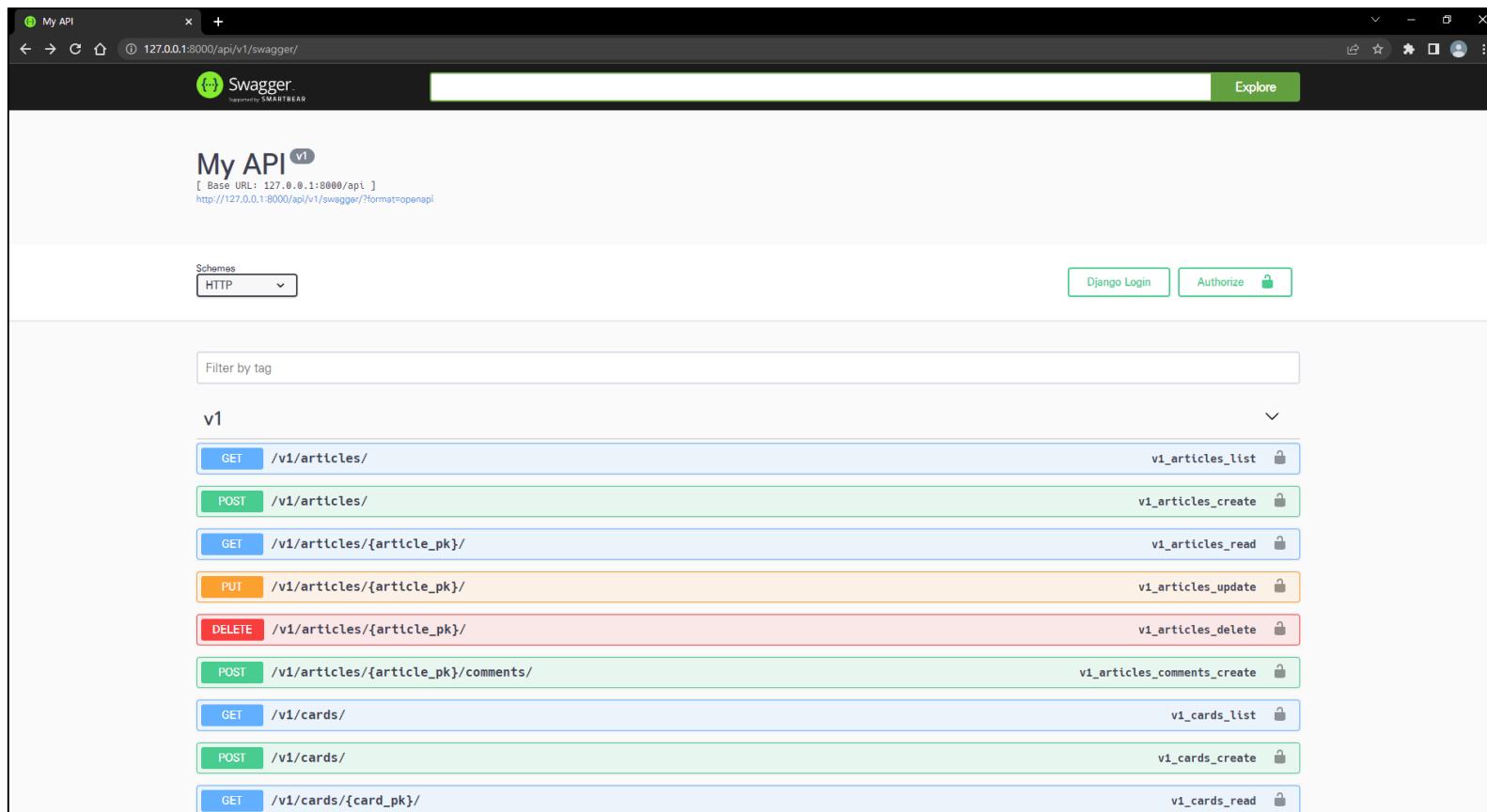
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
from django.urls import path
from . import views

schema_view = get_schema_view(
    openapi.Info(
        title='My API',
        default_version='v1',
        # 아래는 선택 인자입니다.
        description="API 서비스입니다.",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="edujunho.hphk@gmail.com"),
        license=openapi.License(name="SSAFY License"),
    ),
)

urlpatterns = [
    ...
    path('swagger/', schema_view.with_ui('swagger')),
]
```

drf-yasg 라이브러리

- <http://127.0.0.1:8000/api/v1/swagger/> 접속 후 확인



Fixtures

How to provide initial data for models

- 앱을 처음 설정할 때 미리 준비된 데이터로 데이터베이스를 미리 채우는 것이 필요한 상황이 있음
- 마이그레이션 또는 **fixtures**와 함께 초기 데이터를 제공

fixtures

- 데이터베이스의 serialized 된 내용을 포함하는 파일 모음
- django가 fixtures 파일을 찾는 경로
 - app/fixtures/

dumpdata

- 응용 프로그램과 관련된 데이터베이스의 모든 데이터를 표준 출력으로 출력

```
$ python manage.py dumpdata [app_label[.modelName] [app_label[.modelName] ...]]
```

```
# --indent 옵션을 주지 않으면 한 줄로 작성되기 때문에 다음과 같이 설정 가능  
# "auth 앱의 user 모델 데이터를 indent 4칸의 user.json 파일로 출력"
```

```
$ python manage.py dumpdata --indent 4 auth.user > user.json
```

loaddata

- fixture의 내용을 검색하여 데이터베이스로 로드

```
# app/fixtures/user.json 파일을 데이터베이스로 로드  
$ python manage.py loaddata user.json
```

fixtures 실습

- 99_fixtures 프로젝트 준비
- 가상환경 생성 및 활성화
- 패키지 설치 및 migrate

fixtures 실습

- fixtures 데이터 추출을 위해 django-seed 라이브러리를 사용해 데이터 생성

```
$ python manage.py seed articles --number=10
```

fixtures 실습

- 각 모델 별 dumpdata 실행

```
$ python manage.py dumpdata --indent 4 articles.article > article.json  
$ python manage.py dumpdata --indent 4 articles.comment > comments.json  
$ python manage.py dumpdata --indent 4 accounts.user > users.json
```

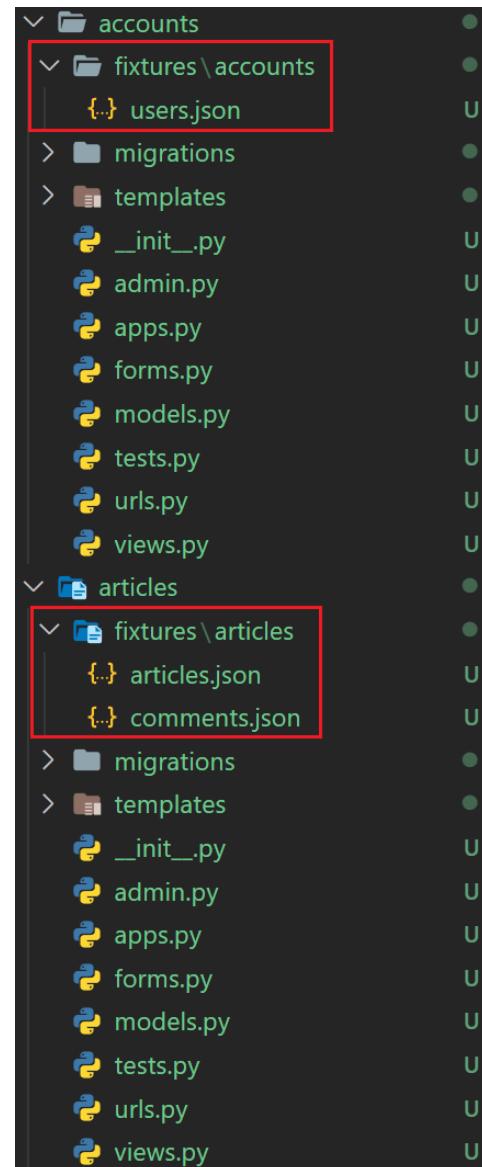
- 실행 결과

>	accounts	•
>	articles	•
>	crud	•
>	templates	•
{..}	articles.json	U
{..}	comments.json	U
⌚	db.sqlite3	
🐍	manage.py	U
📄	requirements.txt	U
{..}	users.json	U

fixtures 실습

- articles/fixtures/articles,
accounts/fixtures/accounts

경로에 추출한 파일 배치하기



fixtures 실습

- loaddata 실행
 - 단, loaddata 전에 데이터베이스 삭제

```
$ python manage.py loaddata articles/articles.json articles/comments.json accounts/users.json  
Installed 30 object(s) from 3 fixture(s)
```

[주의] fixtures는 직접 생성하는 것이 아닌 dumpdata를 통해 생성하는 것이니 직접 작성하려 하지 말 것

Improve query

쿼리셋 이해하기

QuerySets are lazy (1/3)

- “쿼리셋은 게으르다.”
- 쿼리셋을 만드는 작업에는 데이터베이스 작업이 포함되지 않음
- 하루 종일 필터를 함께 쌓을 수 있으며(stack filters),
Django는 쿼리셋이 ‘평가(evaluated)’ 될 때까지 실제로 쿼리를 실행하지 않음
- DB에 쿼리를 전달하는 일이 웹 애플리케이션을 느려지게 하는 주범 중 하나이기 때문

QuerySets are lazy (2/3)

- 다음 구문에서 몇 개의 쿼리가 DB에 전달될까?

```
articles = Article.objects.filter(title__startswith='What')
articles = articles.filter(created_at__lte=datetime.date.today())
articles = articles.exclude(content__icontains='food')
print(articles)
```

QuerySets are lazy (3/3)

- 다음 구문에서 몇 개의 쿼리가 DB에 전달될까?

```
articles = Article.objects.filter(title__startswith='What')
articles = articles.filter(created_at__lte=datetime.date.today())
articles = articles.exclude(content__icontains='food')
print(articles)
```

print(articles)에서 단 **한 번** 전달 됨

| 평가 (evaluated)

- 쿼리셋에 해당하는 DB의 레코드들을 실제로 가져오는 것
 - == hit, access, Queries database
- 평가된 모델들은 쿼리셋의 내장 캐시(cache)에 저장되며,
덕분에 우리가 쿼리셋을 다시 순회하더라도 똑같은 쿼리를 DB에 다시 전달하지 않음

[참고] 캐시 (cache)

- 데이터나 값을 미리 복사해 놓는 임시 장소
- 캐시의 접근 시간에 비해 “원래 데이터를 접근하는 시간이 오래 걸리는 경우” 또는 “값을 다시 계산하는 시간을 절약하고 싶은 경우”에 사용
- 캐시에 데이터를 미리 복사해 놓으면 계산이나 접근 시간 없이
- 더 빠른 속도로 데이터에 접근할 수 있다
- 시스템의 효율성을 위해 여러 분야에서 두루 사용됨

쿼리셋이 평가되는 시점 (When QuerySets are evaluated) (1/2)

1. Iteration

- QuerySet은 반복 가능하며 처음 반복 할 때 데이터베이스 쿼리를 실행

```
for article in Article.objects.all():
    print(article.title)
```

2. bool()

- bool() 또는 if 문 사용과 같은 bool 컨텍스트에서 QuerySet을 테스트하면 쿼리가 실행

```
if Article.objects.filter(title='Test'):
    print('hello')
```

*결과가 하나 이상 존재하는지 확인하기만 한다면 exist()를 사용하는 것이 더 효율적

쿼리셋이 평가되는 시점 (When QuerySets are evaluated) (2/2)

3. 이외 Pickling/Caching, Slicing, `repr()`, `len()`,
`list()`에서 평가됨

캐시와 쿼리셋 (1/2)

- 각 쿼리셋에는 데이터베이스 액세스를 최소화하는 ‘캐시’가 포함 되어있음
 - 새로운 쿼리셋이 만들어지면 캐시는 비어있음
 - 쿼리셋이 처음으로 평가되면 데이터베이스 쿼리가 발생
 - Django는 쿼리 결과를 쿼리셋의 캐시에 저장하고 명시적으로 요청 된 결과를 반환
 - 이후 쿼리셋 평가는 캐시 된 결과를 재사용

캐시와 쿼리셋 (2/2)

```
# 나쁜 예 (동일한 데이터베이스 쿼리가 두 번 실행)
print([article.title for article in Article.objects.all()]) # 평가
print([article.content for article in Article.objects.all()]) # 평가

# 좋은 예
queryset = Article.objects.all()
print([article.title for article in queryset]) # 평가
print([article.content for article in queryset]) # 캐시 재사용
```

쿼리셋이 캐시되지 않는 경우

- 쿼리셋 객체에서 특정 인덱스를 반복적으로 가져오면 매번 데이터베이스를 쿼리

```
queryset = Article.objects.all()  
print(queryset[5]) # Queries the database  
print(queryset[5]) # Queries the database again
```

- 그러나 쿼리셋 전체가 이미 평가 된 경우 캐시에서 확인

```
[article for article in queryset] # Queries the database  
print(queryset[5]) # Uses cache  
print(queryset[5]) # Uses cache
```

쿼리셋 캐시 관련

1. **with** 템플릿 태그 사용하기

- 쿼리셋의 캐싱 동작을 사용하여 더 간단한 이름으로 복잡한 변수를 캐시

```
{% with followers=person.followers.all followings=person.followings.all %}  
  <div>  
    팔로워 : {{ followers|length }} / 팔로우 : {{ followings|length }}  
  </div>  
{% endwith %}
```

2. **iterator()** 사용하기

- 객체가 많을 때 쿼리셋의 캐싱 동작으로 인해 많은 양의 메모리가 사용될 때 사용
- 다음 챕터 like(좋아요) 코드 예시에서 확인할 예정

필요하지 않은 것을
검색하지 않기

Don't retrieve things you don't need

- **.count()**
 - 카운트만 원하는 경우
 - `len(queryset)` 대신 `QuerySet.count()` 사용하기
- **.exists()**
 - 최소한 하나의 결과가 존재하는지 확인하려는 경우
 - `if queryset` 대신 `QuerySet.exists()` 사용하기

| 좋아요 코드 예시

- if 문 때문에 쿼리셋이 ‘평가’ 되고, 이에 따라 쿼리셋 캐시에도 전체 레코드가 저장

```
like_set = article.like_users.filter(pk=request.user.pk)

# if 문은 쿼리셋을 평가한다 -> 캐시가 생김
if like_set:
    # 쿼리셋의 전체 결과가 필요하지는 않은 상황임에도
    # ORM은 전체 결과를 가져온다
    article.like_users.remove(request.user)
```

| 좋아요 코드 예시

- **exists()**는 쿼리셋 캐시를 만들지 않으면서 특정 레코드가 존재하는지 검사
- 결과 전체가 필요하지 않은 경우 유용

```
like_set = article.like_users.filter(pk=request.user.pk)
```

```
# exists()는 쿼리셋 캐시를 만들지 않으면서 레코드가 존재하는지 검사
if like_set.exists():

    # DB에서 가져온 레코드가 하나도 없다면
    # 트래픽과 메모리를 절약할 수 있다

    article.like_users.remove(request.user)
```

| 좋아요 코드 예시

- if문 안에서 반복문이 있다면, 순회할 때는 if문에서 캐시된 쿼리셋이 사용됨

```
like_set = article.like_users.filter(pk=request.user.pk)

if like_set:
    for user in like_set:
        print(user.username)
```

| 좋아요 코드 예시

- if문 안에서 반복문이 있다면, 순회할 때는 if문에서 캐시된 쿼리셋이 사용됨

```
like_set = article.like_users.filter(pk=request.user.pk)

if like_set:
    for user in like_set:
        print(user.username)
```

그런데 쿼리셋이 엄청나게 크다면..? 쿼리셋 캐시 자체가 문제가 될 수 있음

| 좋아요 코드 예시

- **iterator()**는 객체가 많을 때 쿼리셋의 캐싱 동작으로 인해 많은 양의 메모리가 사용될 때 사용
- 몇 천 개 단위의 레코드를 다뤄야 할 경우,
이 데이터를 한 번에 가져와 메모리에 올리는 행위는 매우 비효율적이기 때문
- 데이터를 작은 덩어리로 쪼개어 가져오고, 이미 사용한 레코드는 메모리에서 지움

```
like_set = article.like_users.filter(pk=request.user.pk)

if like_set:
    for user in like_set.iterator():
        print(user.username)
```

| 좋아요 코드 예시

- 그런데 쿼리셋이 엄청 큰 경우 평가되는 if 문도 문제가 될 수 있음

```
like_set = article.like_users.filter(pk=request.user.pk)

# 첫 번째 쿼리로, 쿼리셋에 레코드가 존재하는지 확인
if like_set.exists():

    # 또다른 쿼리로 레코드를 조금씩 가져옴
    for user in like_set.iterator():

        print(user.username)
```

“안일한 최적화 주의”

- **exists()**와 **iterator()** 메서드를 사용하면 메모리 사용을 최적화할 수 있지만, 쿼리셋 캐시는 생성되지 않기 때문에, DB 쿼리가 중복될 수 있음

Annotate

실습 준비

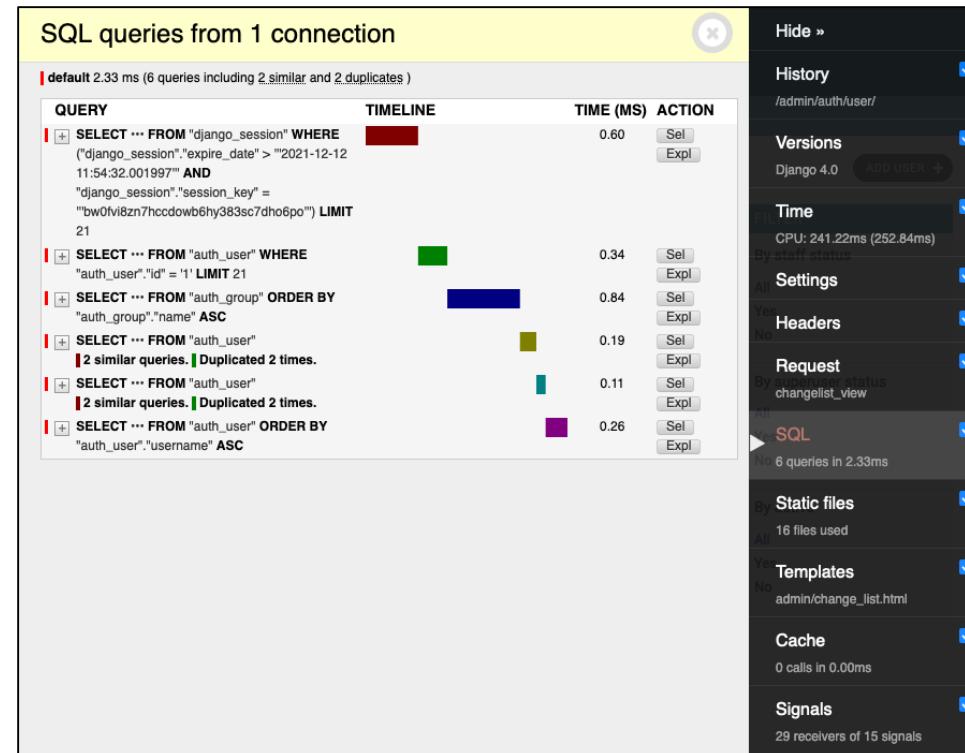
- 99_improve_query 프로젝트 준비
- 가상환경 생성 및 활성화
- 패키지 설치
- migrate 및 fixture 데이터 load

```
$ python manage.py migrate
```

```
$ python manage.py loaddata users.json articles.json comments.json
```

[참고] Django Debug Toolbar

- 현재 요청/응답에 대한 다양한 디버그 정보를 표시하고,
다양한 패널에서 자세한 정보를 표시



annotate (1/7)

- 단순히 SQL로 계산해 하나의 테이블의 필드로 추가하여 붙여 올 수 있는 경우
- “게시글 별로 댓글 수를 출력 해보기”

annotate (2/7)

- 개선 전

```
# articles/index_1.html

def index_1(request):
    articles = Article.objects.order_by( '-pk' )
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_1.html', context)
```

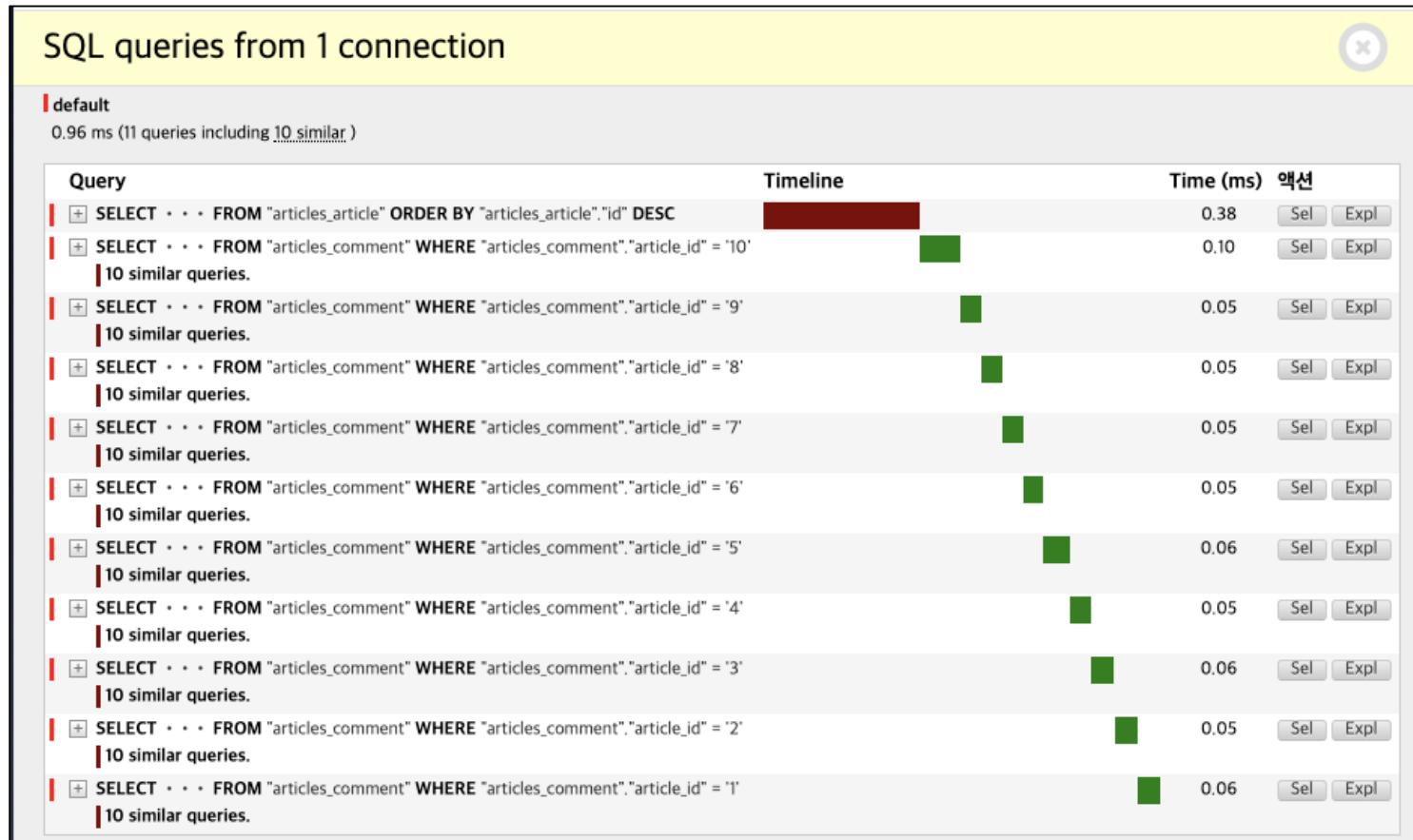
```
<!-- articles/index_1.html -->

{% extends 'base.html' %}

{% block content %}
    <h1>Articles</h1>
    {% for article in articles %}
        <p>제목 : {{ article.title }}</p>
        <p>댓글개수 : {{ article.comment_set.count }}</p>
        <hr>
    {% endfor %}
    {% endblock content %}
```

annotate (3/7)

- 개선 전 – 11 queries including 10 similar



annotate (4/7)

- 개선 전 – SQL 구문

```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

-- 아래 구문 10번 반복
SELECT COUNT(*) AS "__count"
  FROM "articles_comment"
 WHERE "articles_comment"."article_id" = '10'
```

annotate (5/7)

- 개선 후

```
# articles/views.py

from django.db.models import Count

def index_1(request):
    articles = Article.objects.annotate(Count('comment')).order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_1.html', context)
```

```
<!-- articles/index_1.html -->

{% extends 'base.html' %}

{% block content %}
    <h1>Articles</h1>
    {% for article in articles %}
        <p>제목 : {{ article.title }}</p>
        <p>댓글개수 : {{ article.comment__count }}</p>
        <hr>
    {% endfor %}
{% endblock content %}
```

annotate (6/7)

- 개선 후 – 11 queries including 10 similar → 1 query



annotate (7/7)

- 개선 후 – SQL 구문

```
SELECT
    "articles_article"."id",
    "articles_article"."user_id",
    "articles_article"."title",
    "articles_article"."content",
    "articles_article"."created_at",
    "articles_article"."updated_at",
    COUNT("articles_comment"."id") AS "comment__count"
FROM "articles_article"
LEFT OUTER JOIN "articles_comment" ON ("articles_article"."id" = "articles_comment"."article_id")
GROUP BY
    "articles_article"."id",
    "articles_article"."user_id",
    "articles_article"."title",
    "articles_article"."content",
    "articles_article"."created_at",
    "articles_article"."updated_at"
ORDER BY "articles_article"."id" DESC
```

[참고] JOIN 개요

- 두 개 이상의 테이블들을 연결 또는 결합하여 데이터를 출력하는 것을 JOIN이라고 함
- 관계형 데이터베이스의 가장 큰 장점이자 핵심적인 기능
- 일반적으로 PK 나 FK 값의 연관에 의해 JOIN이 성립
- SQL JOIN 에 대해서는 다양한 Visualization 사이트 참고하기
 - <https://sql-joins.leopard.in.ua/>
 - <https://joins.spathon.com/>

| 이제부터 시작

- 여기까지는 중복을 제거하지 않고 단순히 쿼리 개수만 날린 것
- 이것보다 더 큰 문제는 반복문을 도는 상황에서의 1:N, M:N 호출상황이다.

한번에 모든 것을 검색하기

Retrieve everything at once if you know you will need it

1. `select_related()`

- 1:1 또는 1:N 참조 관계에서 사용
- DB에서 INNER JOIN을 활용

2. `prefetch_related()`

- M:N 또는 1:N 역참조 관계에서 사용
- DB가 아닌 Python 을 통한 JOIN

| `select_related()` (1/6)

- SQL의 INNER JOIN을 실행하여 테이블의 일부를 가져오고,
SELECT FROM에서 관련된 필드들을 가져옴
- 단, single-valued relationships 관계(foreign key and one-to-one)에서만
사용 가능
- “게시글의 사용자 이름까지 출력을 해보기”

| select_related() (2/6)

- 개선 전

```
# articles/views.py

def index_2(request):
    articles = Article.objects.order_by(' -pk ')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_2.html', context)
```

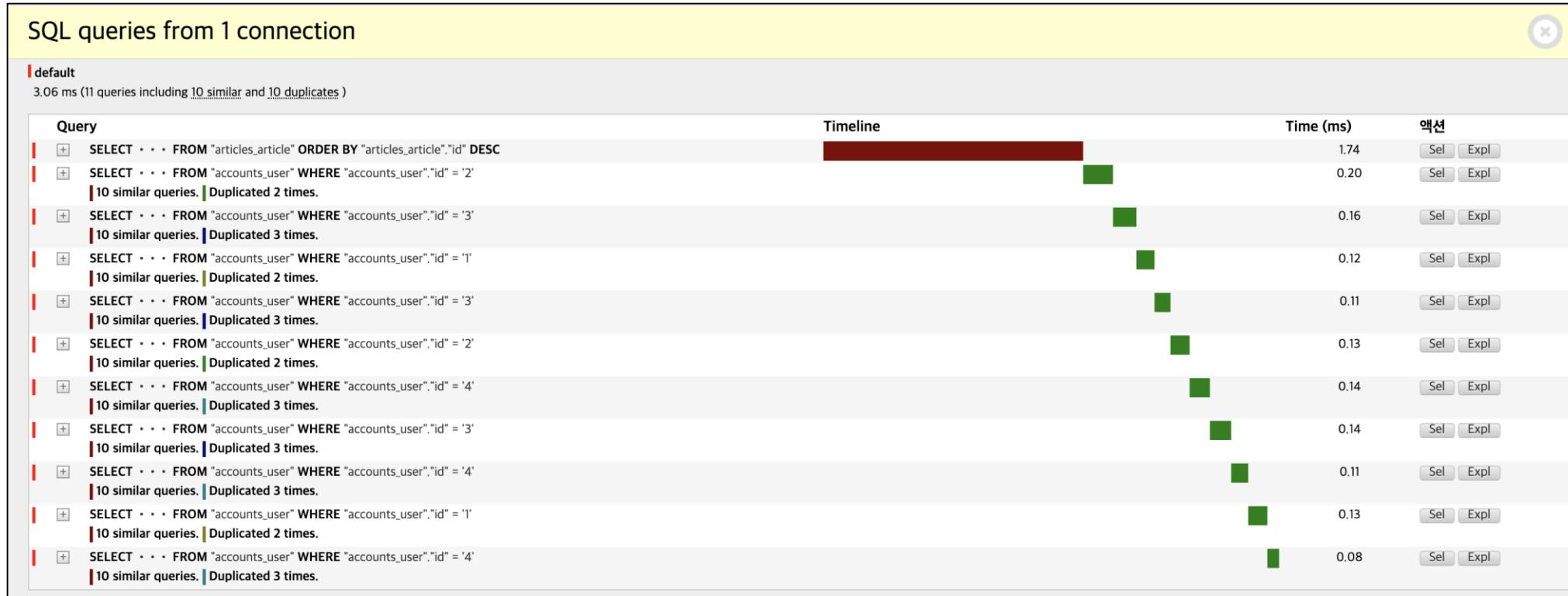
```
<!-- articles/index_2.html -->

{% extends 'base.html' %}

{% block content %}
<h1>Articles</h1>
{% for article in articles %}
<h3>작성자 : {{ article.user.username }}</h3>
<p>제목 : {{ article.title }}</p>
<hr>
{% endfor %}
{% endblock content %}
```

| select_related() (3/6)

- 개선 전 – 11 queries including 10 similar and 10 duplicates



| select_related() (4/6)

- 개선 후

```
# articles/views.py

def index_2(request):
    articles = Article.objects.select_related('user').order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_2.html', context)
```

| select_related() (5/6)

- 개선 후 – 11 queries including 10 similar and 10 duplicates → 1 query

SQL queries from 1 connection

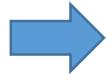
Query	Timeline	Time (ms)	액션
<code>+ SELECT ... FROM "articles_article" INNER JOIN "accounts_user" ON ("articles_article"."user_id" = "accounts_user"."id") ORDER BY "articles_article"."id" DESC</code>		0.00	<code>Sel</code> <code>Expl</code>

| select_related() (6/6)

- 개선 후 – SQL 구문

```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

-- 10 similar and 10 duplicates
SELECT "accounts_user"."id",
       "accounts_user"."password",
       "accounts_user"."last_login",
       "accounts_user"."is_superuser",
       "accounts_user"."username",
       "accounts_user"."first_name",
       "accounts_user"."last_name",
       "accounts_user"."email",
       "accounts_user"."is_staff",
       "accounts_user"."is_active",
       "accounts_user"."date_joined"
  FROM "accounts_user"
 WHERE "accounts_user"."id" = '5'
```



```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at",
       "accounts_user"."id",
       "accounts_user"."password",
       "accounts_user"."last_login",
       "accounts_user"."is_superuser",
       "accounts_user"."username",
       "accounts_user"."first_name",
       "accounts_user"."last_name",
       "accounts_user"."email",
       "accounts_user"."is_staff",
       "accounts_user"."is_active",
       "accounts_user"."date_joined"
  FROM "articles_article"
 INNER JOIN "accounts_user" ON ("articles_article"."user_id" = "accounts_user"."id")
 ORDER BY "articles_article"."id" DESC
```

| **prefetch_related() (1/6)**

- selected_related와 달리 SQL의 JOIN을 실행하지 않고, python에서 joining을 실행
- select_related가 지원하는 single-valued relationships 관계에 더해, select_related를 사용하여 수행 할 수 없는 M:N and 1:N 역참조 관계에서 사용 가능
- “댓글 목록을 모두 출력을 해보기”

| prefetch_related() (2/6)

- 개선 전

```
# articles/views.py

def index_3(request):
    articles = Article.objects.order_by( '-pk' )
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_3.html', context)
```

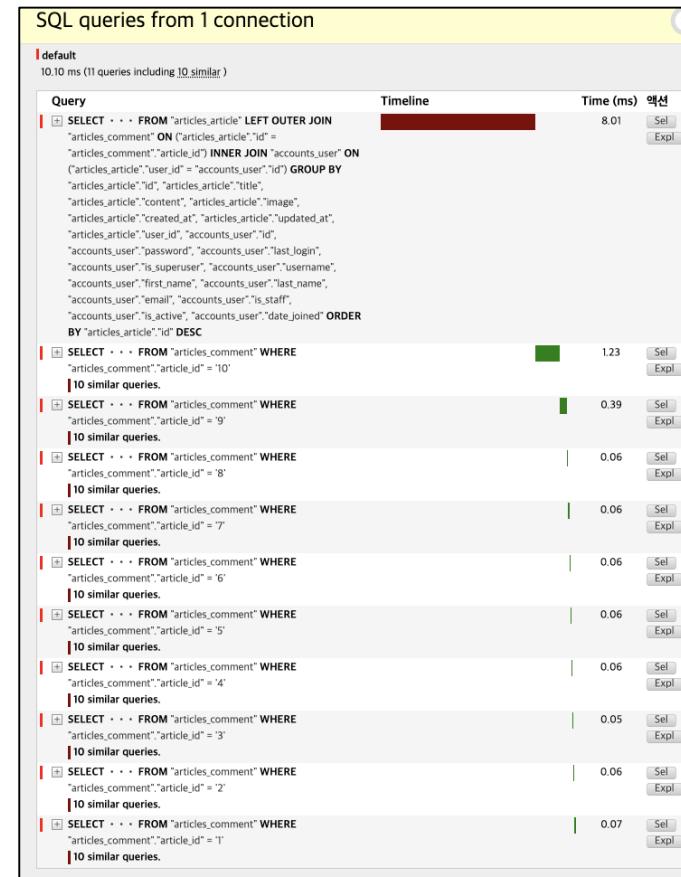
```
<!-- articles/index_3.html -->

{% extends 'base.html' %}

{% block content %}
<h1>Articles</h1>
{% for article in articles %}
<p>제목 : {{ article.title }}</p>
<p>댓글 목록</p>
{% for comment in article.comment_set.all %}
<p>{{ comment.content }}</p>
{% endfor %}
<hr>
{% endfor %}
{% endblock content %}
```

prefetch_related() (3/6)

- 개선 전 – 11 queries including 10 similar



| prefetch_related() (4/6)

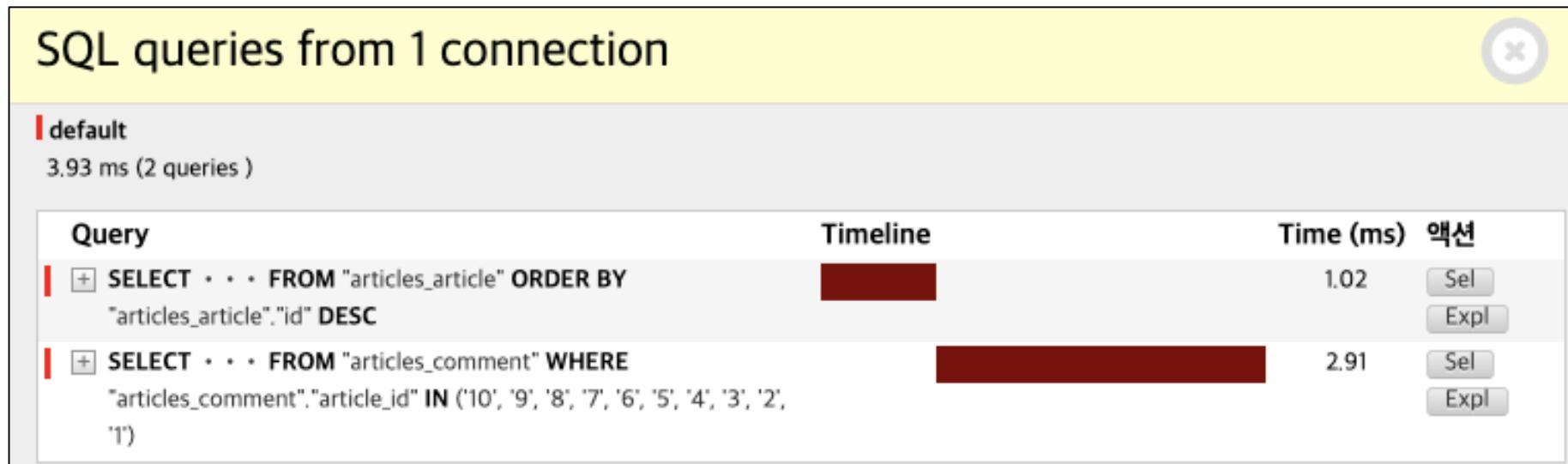
- 개선 후

```
# articles/views.py

def index_3(request):
    articles = Article.objects.prefetch_related('comment_set').order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_3.html', context)
```

| prefetch_related() (5/6)

- 개선 후 – 2 queries

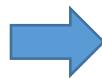


prefetch_related() (6/6)

- 개선 후 – SQL 구문

```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

-- 10 similar queries
SELECT "articles_comment"."id",
       "articles_comment"."article_id",
       "articles_comment"."user_id",
       "articles_comment"."content",
       "articles_comment"."created_at",
       "articles_comment"."updated_at"
  FROM "articles_comment"
 WHERE "articles_comment"."article_id" = '10'
```



```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

SELECT "articles_comment"."id",
       "articles_comment"."article_id",
       "articles_comment"."user_id",
       "articles_comment"."content",
       "articles_comment"."created_at",
       "articles_comment"."updated_at"
  FROM "articles_comment"
 WHERE "articles_comment"."article_id" IN ('10', '9', '8', '7', '6', '5', '4', '3', '2', '1')
```

| 복합 활용 (1/9)

- “댓글에 더해서 해당 댓글을 작성한 사용자 이름까지 출력 해보기”

복합 활용 (2/9)

- 개선 전

```
# articles/views.py

from django.db.models import Prefetch

def index_4(request):
    articles = Article.objects.order_by( '-pk' )
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_4.html', context)
```

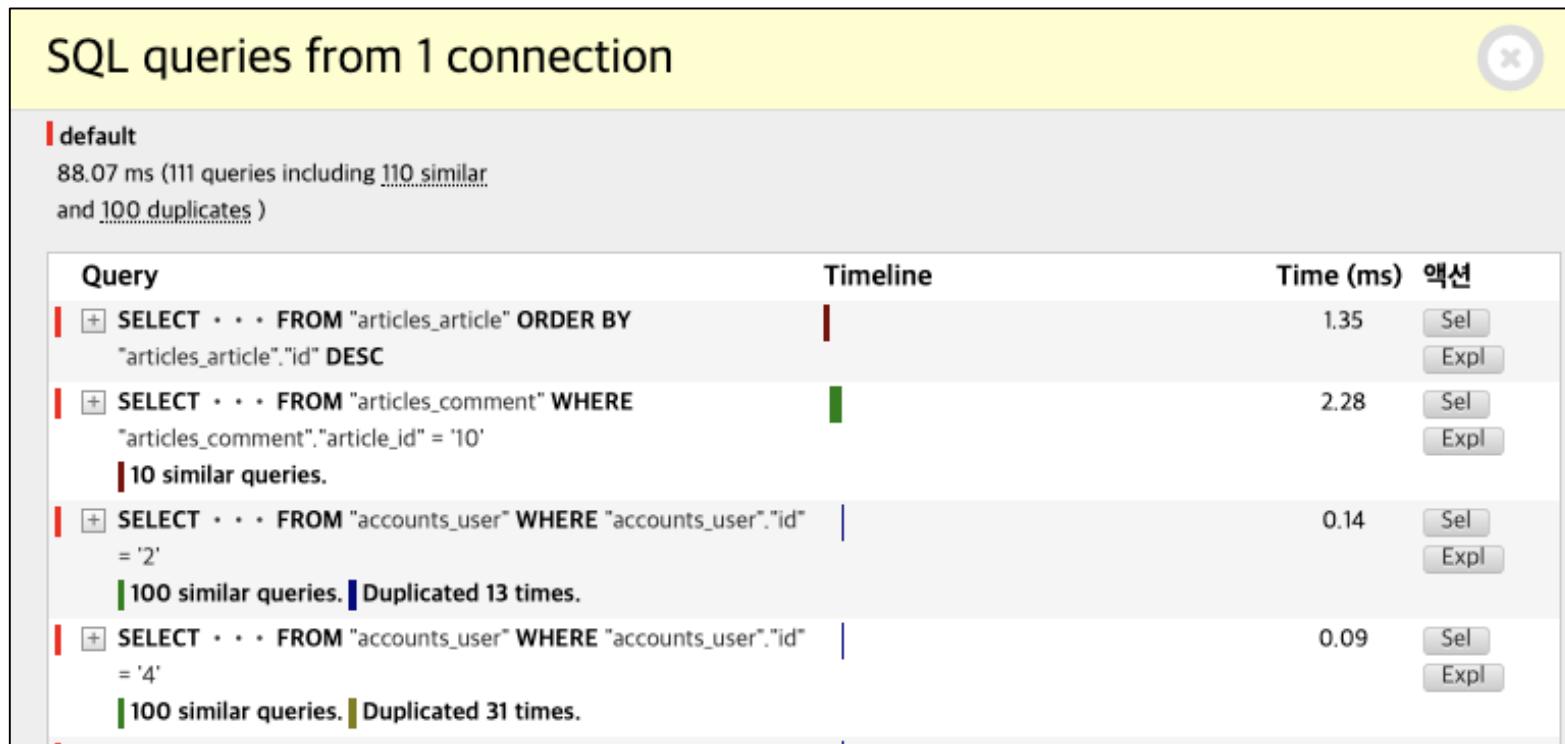
```
<!-- articles/index_4.html -->

{% extends 'base.html' %}

{% block content %}
<h1>Articles</h1>
{% for article in articles %}
<p>제목 : {{ article.title }}</p>
<p>댓글 목록</p>
{% for comment in article.comment_set.all %}
<p>{{ comment.user.username }} : {{ comment.content }}</p>
{% endfor %}
<hr>
{% endfor %}
{% endblock content %}
```

복합 활용 (3/9)

- 개선 전 – 111 queries including 110 similar and 100 duplicates
- 1개 + 10개 + 10개에 대한 10개 = 111개



복합 활용 (4/9)

- 개선 후 1단계

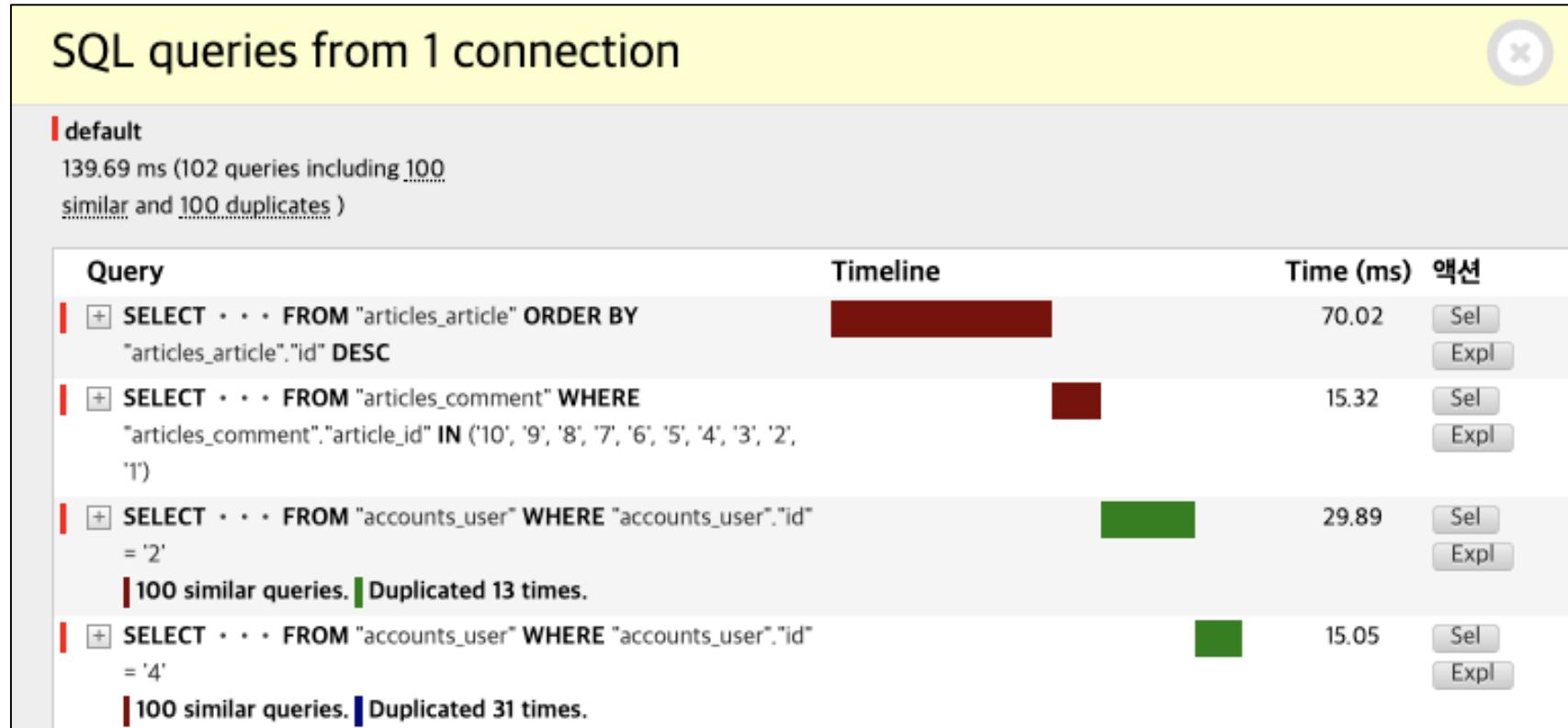
```
# articles/views.py

from django.db.models import Prefetch

def index_4(request):
    articles = Article.objects.prefetch_related('comment_set').order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_4.html', context)
```

복합 활용 (5/9)

- 개선 후 1단계 – 102 queries including 100 similar and 100 duplicates



복합 활용 (6/9)

- 개선 후 1단계 – SQL 구문

```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

-- 10 similar queries.
SELECT "articles_comment"."id",
       "articles_comment"."article_id",
       "articles_comment"."user_id",
       "articles_comment"."content",
       "articles_comment"."created_at",
       "articles_comment"."updated_at"
  FROM "articles_comment"
 WHERE "articles_comment"."article_id" = '10'

-- 100 similar queries.
SELECT "accounts_user"."id",
       "accounts_user"."password",
       "accounts_user"."last_login",
       "accounts_user"."is_superuser",
       "accounts_user"."username",
       "accounts_user"."first_name",
       "accounts_user"."last_name",
       "accounts_user"."email",
       "accounts_user"."is_staff",
       "accounts_user"."is_active",
       "accounts_user"."date_joined"
  FROM "accounts_user"
 WHERE "accounts_user"."id" = '1'
```



```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

SELECT "articles_comment"."id",
       "articles_comment"."article_id",
       "articles_comment"."user_id",
       "articles_comment"."content",
       "articles_comment"."created_at",
       "articles_comment"."updated_at"
  FROM "articles_comment"
 WHERE "articles_comment"."article_id" IN ('10', '9', '8', '7', '6', '5', '4', '3', '2', '1')

-- 100 similar queries.
SELECT "accounts_user"."id",
       "accounts_user"."password",
       "accounts_user"."last_login",
       "accounts_user"."is_superuser",
       "accounts_user"."username",
       "accounts_user"."first_name",
       "accounts_user"."last_name",
       "accounts_user"."email",
       "accounts_user"."is_staff",
       "accounts_user"."is_active",
       "accounts_user"."date_joined"
  FROM "accounts_user"
 WHERE "accounts_user"."id" = '1'
```

| 복합 활용 (7/9)

- 개선 후 2단계

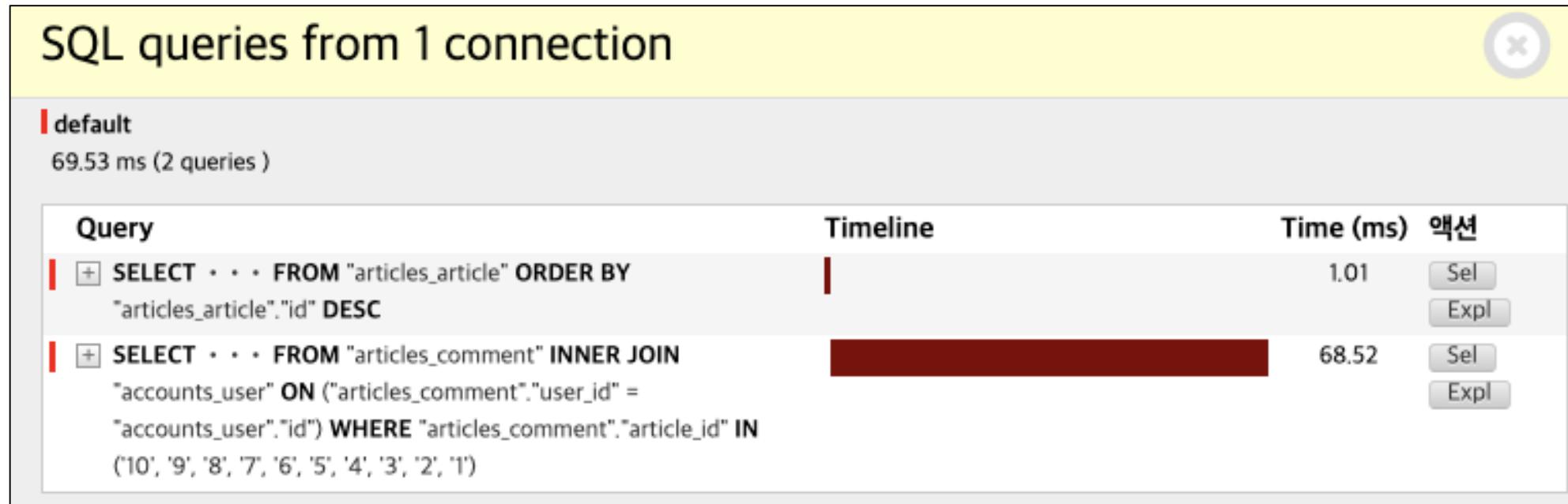
```
# articles/views.py

from django.db.models import Prefetch

def index_4(request):
    articles = Article.objects.prefetch_related(
        Prefetch('comment_set', queryset=Comment.objects.select_related('user'))
    ).order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_4.html', context)
```

복합 활용 (8/9)

- 개선 후 2단계 – 2 queries



복합 활용 (9/9)

- 개선 후 2단계 – SQL 구문

```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

SELECT "articles_comment"."id",
       "articles_comment"."article_id",
       "articles_comment"."user_id",
       "articles_comment"."content",
       "articles_comment"."created_at",
       "articles_comment"."updated_at"
  FROM "articles_comment"
 WHERE "articles_comment"."article_id" IN ('10', '9', '8', '7', '6', '5', '4', '3', '2', '1')

-- 100 similar queries.
SELECT "accounts_user"."id",
       "accounts_user"."password",
       "accounts_user"."last_login",
       "accounts_user"."is_superuser",
       "accounts_user"."username",
       "accounts_user"."first_name",
       "accounts_user"."last_name",
       "accounts_user"."email",
       "accounts_user"."is_staff",
       "accounts_user"."is_active",
       "accounts_user"."date_joined"
  FROM "accounts_user"
 WHERE "accounts_user"."id" = '1'
```



```
SELECT "articles_article"."id",
       "articles_article"."user_id",
       "articles_article"."title",
       "articles_article"."content",
       "articles_article"."created_at",
       "articles_article"."updated_at"
  FROM "articles_article"
 ORDER BY "articles_article"."id" DESC

SELECT "articles_comment"."id",
       "articles_comment"."article_id",
       "articles_comment"."user_id",
       "articles_comment"."content",
       "articles_comment"."created_at",
       "articles_comment"."updated_at",
       "accounts_user"."id",
       "accounts_user"."password",
       "accounts_user"."last_login",
       "accounts_user"."is_superuser",
       "accounts_user"."username",
       "accounts_user"."first_name",
       "accounts_user"."last_name",
       "accounts_user"."email",
       "accounts_user"."is_staff",
       "accounts_user"."is_active",
       "accounts_user"."date_joined"
  FROM "articles_comment"
 INNER JOIN "accounts_user" ON ("articles_comment"."user_id" = "accounts_user"."id")
 WHERE "articles_comment"."article_id" IN ('10', '9', '8', '7', '6', '5', '4', '3', '2', '1')
```

섣부른 최적화

"작은 효율성(small efficiency)에 대해서는,
말하자면 97% 정도에 대해서는, 잊어버려라.
섣부른 최적화(premature optimization)는 모든 악의 근원이다."

– Donald E. Knuth

“The Art of Computer Programming” 저자
“알고리즘 분석” 분야 창시자
조판 프로그램 “TeX” 창시자

마무리

감사합니다 !