Django Admin at Scale: From Milliseconds to Microseconds 

Sumit
DjangoCon Europe 2025/Dublin

The Django Admin Dream

- **Built-in CRUD operations without writing a line of code**
- **Automatic form generation and validation**
- **Authentication and permissions handling**
- **Customizable interface**

"The best thing since sliced bread" for rapid development

A Real Story

Last year, I consulted for a streaming company where we had a critical table called, `NotificationLog`.

This table tracked:

- Email templates used
- When emails were opened/clicked
- Campaign performance data
- User engagement metrics

Critical for the business - our marketing/product team relied on this data.

Then Disaster Struck ….

- **Our product manager needed to check how new campaigns were performing**
- **Admin page started returning 504 timeout errors**
- **Frustrated PM**
- **Marketing campaigns were flying blind**

"We need these pages working again."

The Business Impact

- **Product managers couldn't verify campaign performance**
- **Marketing decisions were delayed**
- **New features were put on hold while we fixed admin**
- **Team credibility was damaged**
- **Actual revenue impact from delayed campaign optimizations**

The Turnaround

Before:

- **Timeout errors (>30 seconds)**
- **Frustrated teams**
- **Business Impact**

After:

- **<200ms response time**
- **Happy product managers**
- **Data-driven marketing decisions**

Today’s Journey

1. **Diagnosis**: Identifying common performance bottlenecks
2. **Solutions**: Battle-tested optimization techniques
3. **Maintenance**: Keeping performance gains over time

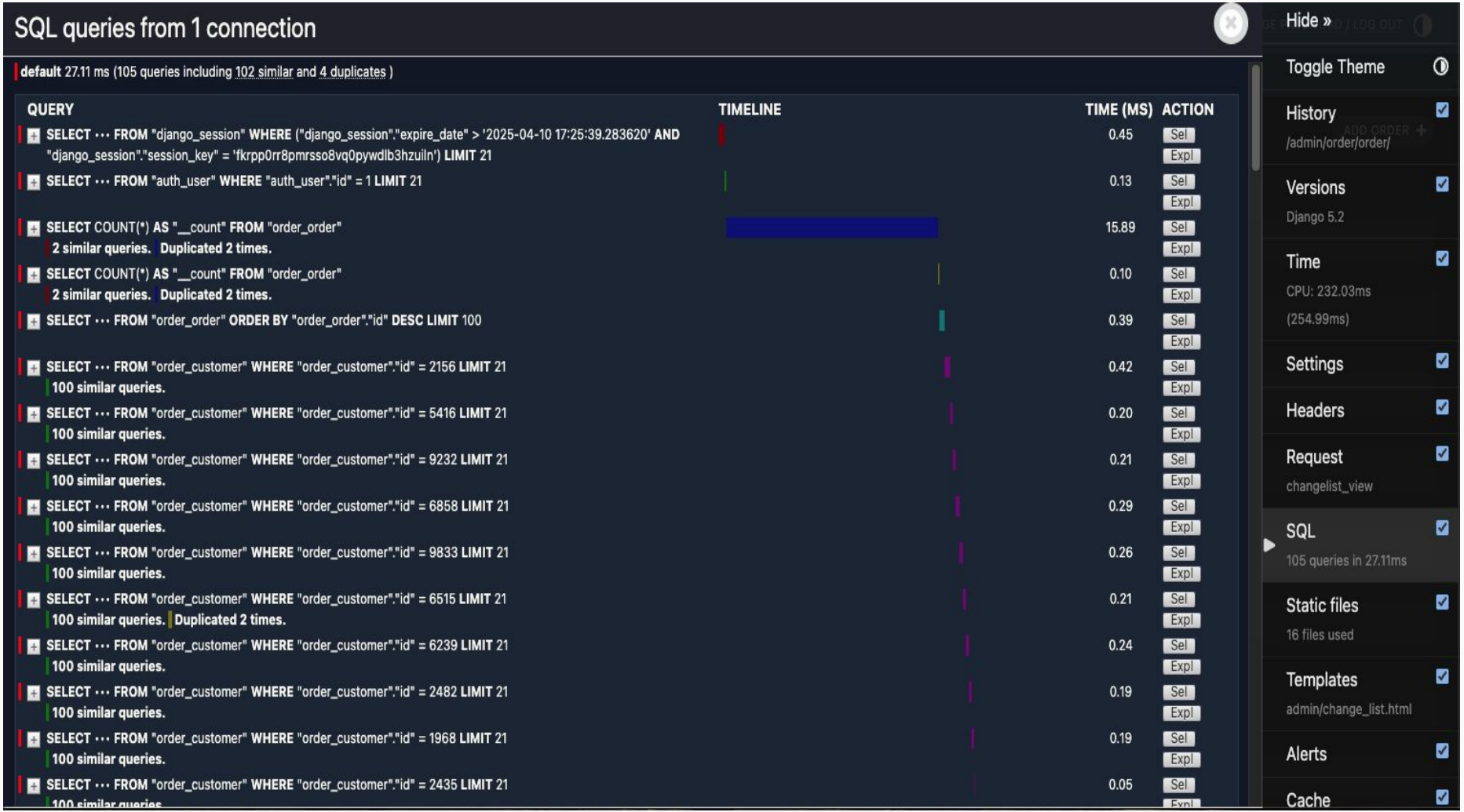
Part 1: Diagnosis

Understanding What Makes Django Admin Slow

Common Bottleneck #1: N+1 Queries

```
# A seemingly innocent ModelAdmin
class OrderAdmin(admin.ModelAdmin):
    list_display = ('id', 'customer_name', 'total', 'status')

# But behind the scenes...
# - 1 query to fetch Orders
# - N queries to fetch related Customer for each Order
# = N+1 queries 😱
```


SQL queries from 1 connection
default 27.11 ms (105 queries including 102 similar and 4 duplicates)
QUERY
TIMELINE
TIME (MS)
ACTION
SELECT ··· FROM "django_session" WHERE ("django_session"."expire_date" > '2025-04-10 17:25:39.283620' AND "django_session"."session_key" = 'fkrpp0rr8pmrsso8vq0pywdlb3hzuiln') LIMIT 21
0.45
Sel
Expl
SELECT ··· FROM "auth_user" WHERE "auth_user"."id" = 1 LIMIT 21
0.13
Sel
Expl
SELECT COUNT(*) AS "__count" FROM "order_order"
2 similar queries. Duplicated 2 times.
15.89
Sel
Expl
SELECT COUNT(*) AS "__count" FROM "order_order"
2 similar queries. Duplicated 2 times.
0.10
Sel
Expl
SELECT ··· FROM "order_order" ORDER BY "order_order"."id" DESC LIMIT 100
0.39
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 2156 LIMIT 21
100 similar queries.
0.42
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 5416 LIMIT 21
100 similar queries.
0.20
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 9232 LIMIT 21
100 similar queries.
0.21
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 6858 LIMIT 21
100 similar queries.
0.29
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 9833 LIMIT 21
100 similar queries.
0.26
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 6515 LIMIT 21
100 similar queries. Duplicated 2 times.
0.21
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 6239 LIMIT 21
100 similar queries.
0.24
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 2482 LIMIT 21
100 similar queries.
0.19
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 1968 LIMIT 21
100 similar queries.
0.19
Sel
Expl
SELECT ··· FROM "order_customer" WHERE "order_customer"."id" = 2435 LIMIT 21
0.05
Sel
Hide »
Toggle Theme
History
/admin/order/order/
Versions
Django 5.2
Time
CPU: 232.03ms
(254.99ms)
Settings
Headers
Request
changelist_view
SQL
105 queries in 27.11ms
Static files
16 files used
Templates
admin/change_list.html
Alerts
Cache

Common Bottleneck #2: Inefficient Filtering

```
# Looks harmless
class ProductAdmin(admin.ModelAdmin):
    list_filter = ('category', 'tags', 'in_stock')

# But with millions of records...
# - Each filter option requires full table scans
# - Combined filters multiply the problem
```

Common Bottleneck #3: Memory-Intensive Admin Actions

```python
# Default admin actions or custom ones
class ProductAdmin(admin.ModelAdmin):
    actions = ['mark_as_featured', 'recalculate_inventory']

    def mark_as_featured(self, request, queryset):
        # Django loads ALL selected objects into memory first
        for product in queryset:  # Potential OOM with thousands selected
            product.is_featured = True
            product.save()  # Individual saves = N queries
```

Common Bottleneck #4: Expensive Change Forms

```python
class ProductAdmin(admin.ModelAdmin):
    # ForeignKey fields with thousands of options
    # Many-to-many fields with no limits
    fields = ('name', 'category', 'tags', 'related_products')

    # Django loads ALL options for each field
    # - All categories (could be thousands)
    # - All tags (could be tens of thousands)
    # - All other products for 'related_products'
```

Diagnosing Your Own Admin

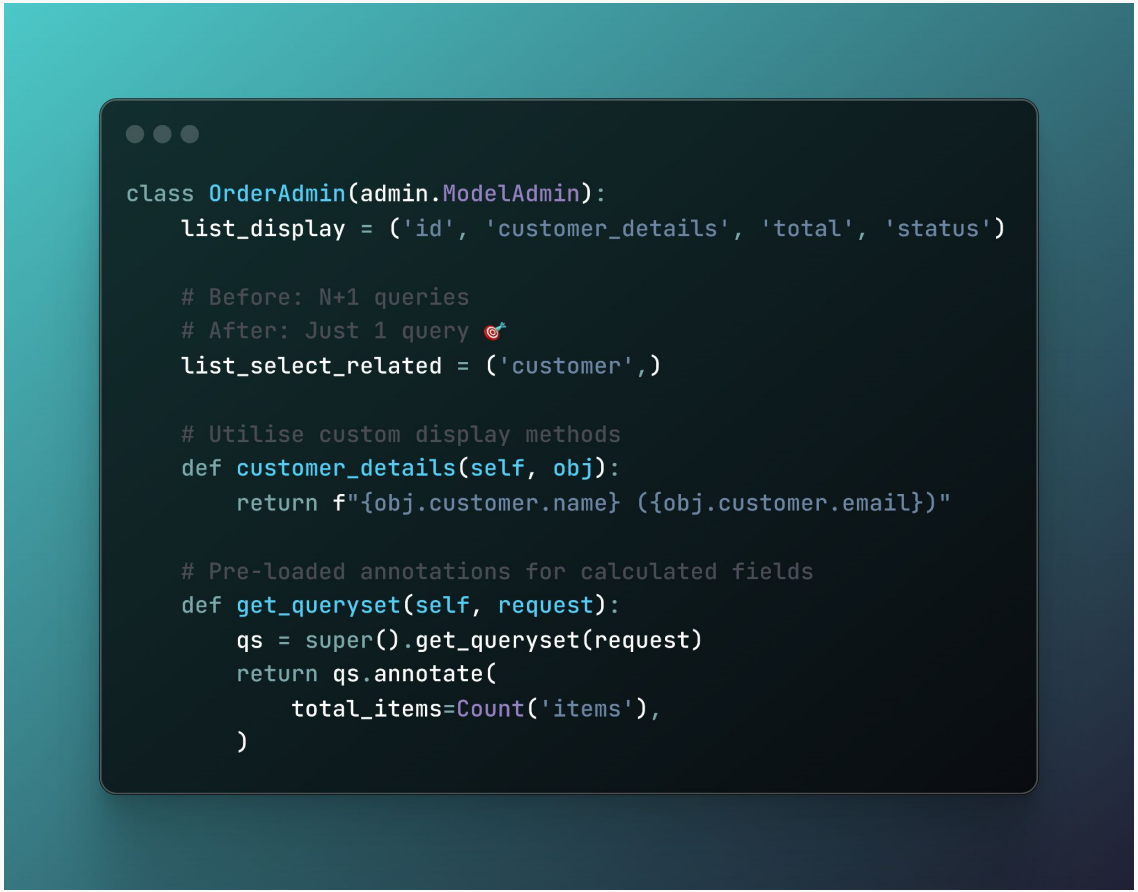
Tools to identify bottlenecks:

- **Django Debug Toolbar**
- **Database query logs**
- **Django Silk Profiler**
- **Queryset.explain()**
- **New Relic/Datadog APM/Sentry APM**

Part 2: Solutions

Transforming Performance

Solution #1: Strategic Related Field Loading



```python
class OrderAdmin(admin.ModelAdmin):
    list_display = ('id', 'customer_details', 'total', 'status')

    # Before: N+1 queries
    # After: Just 1 query 🎯
    list_select_related = ('customer',)

    # Utilise custom display methods
    def customer_details(self, obj):
        return f"{obj.customer.name} ({obj.customer.email})"

    # Pre-loaded annotations for calculated fields
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        return qs.annotate(
            total_items=Count('items'),
        )
```

Solution #2: Advanced Pagination

```python
class MillionRecordAdmin(admin.ModelAdmin):
    # Default pagination is too small
    list_per_page = 100  # Careful with this!

    # Better approach - keyset pagination
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        return KeysetPaginatedQuerySet(qs)  # Custom implementation
```

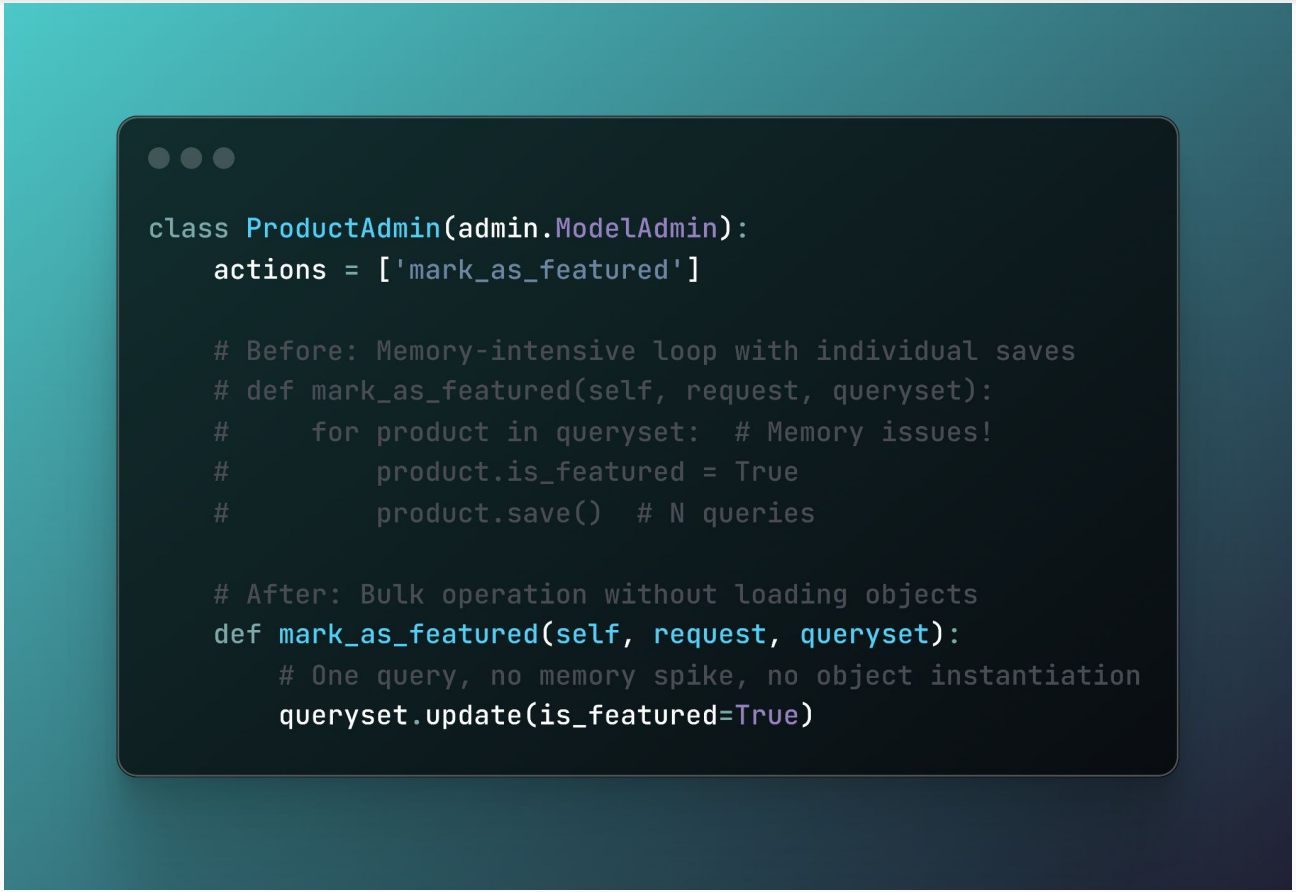
Keyset Pagination Implementation

```python
class KeysetPaginatedQuerySet:
    """Paginate using the primary key instead of OFFSET/LIMIT"""

    def paginate_queryset(self, queryset, page_num):
        if page_num == 1:
            return queryset.order_by('id')[:100]

        last_id = self.get_last_id_from_previous_page(page_num)
        return queryset.filter(id__gt=last_id).order_by('id')[:100]
```

Solution #3: Optimized Admin Actions



```python
class ProductAdmin(admin.ModelAdmin):
    actions = ['mark_as_featured']

    # Before: Memory-intensive loop with individual saves
    # def mark_as_featured(self, request, queryset):
    #     for product in queryset:  # Memory issues!
    #         product.is_featured = True
    #         product.save()  # N queries

    # After: Bulk operation without loading objects
    def mark_as_featured(self, request, queryset):
        # One query, no memory spike, no object instantiation
        queryset.update(is_featured=True)
```

Batch Processing for Complex Actions

```python
class ProductAdmin(admin.ModelAdmin):
    ...

    # After: Bulk operation without loading objects
    def mark_as_featured(self, request, queryset):
        # For more complex actions that need object processing
        # Process in batches to limit memory usage
        queryset.update(being_processed=True)
        for product_batch in self._batch_qs(queryset, 1000):
            self._process_batch(product_batch)

    def _batch_qs(self, queryset, batch_size=1000):
    """Process queryset in batches to avoid memory issues."""
    offset = 0
    while True:
        batch = queryset[offset:offset+batch_size]
        if not batch:
            break
        yield batch
        offset += batch_size
```

Solution #4: Efficient Change Forms

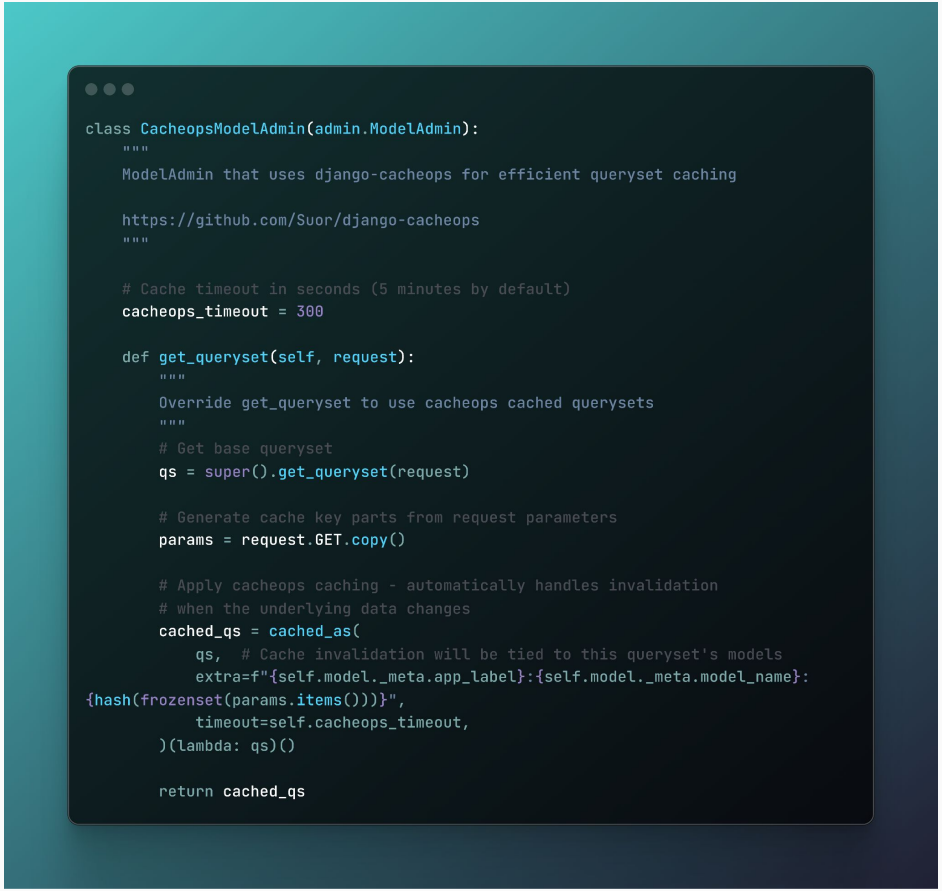
```python
class ProductAdmin(admin.ModelAdmin):
    # Replace massive dropdown menus with search interfaces
    raw_id_fields = ("category", "supplier")

    # Limit initial choices for many-to-many fields
    filter_horizontal = ("tags",)

    # Custom form to control queryset size
    def get_form(self, request, obj=None, **kwargs):
        form = super().get_form(request, obj, **kwargs)
        # Limit related product choices to same category
        if obj:
            form.base_fields["related_products"].queryset = Product.objects.filter(
                category=obj.category
            )[:100]
        return form

    # Defer expensive fields until needed (tabbed interface)
    fieldsets = (
        (
            "Basic",
            {
                "fields": ("name", "price", "description"),
            },
        ),
        (
            "Advanced",
            {
                "fields": ("related_products", "tags"),
                "classes": ("collapse",),  # Initially collapsed
            },
        ),
    )
```

Solution #5: Targeted Caching



```python
class CacheopsModelAdmin(admin.ModelAdmin):
    """
    ModelAdmin that uses django-cacheops for efficient queryset caching

    https://github.com/Suor/django-cacheops
    """

    # Cache timeout in seconds (5 minutes by default)
    cacheops_timeout = 300

    def get_queryset(self, request):
        """
        Override get_queryset to use cacheops cached querysets
        """
        # Get base queryset
        qs = super().get_queryset(request)

        # Generate cache key parts from request parameters
        params = request.GET.copy()

        # Apply cacheops caching - automatically handles invalidation
        # when the underlying data changes
        cached_qs = cached_as(
            qs,  # Cache invalidation will be tied to this queryset's models
            extra=f"{self.model._meta.app_label}:{self.model._meta.model_name}:
{hash(frozenset(params.items()))}",
            timeout=self.cacheops_timeout,
        )(lambda: qs)()

        return cached_qs
```

Using Cacheops for ProductAdmin

```
class ProductAdmin(CacheopsModelAdmin):
    list_display = ('name', 'price', 'category', 'in_stock')
    list_filter = ('category', 'in_stock')
    search_fields = ('name', 'description')

    # Use a shorter cache timeout for products (1 minute)
    cacheops_timeout = 60
```

Solution #6: List Filter Optimizations

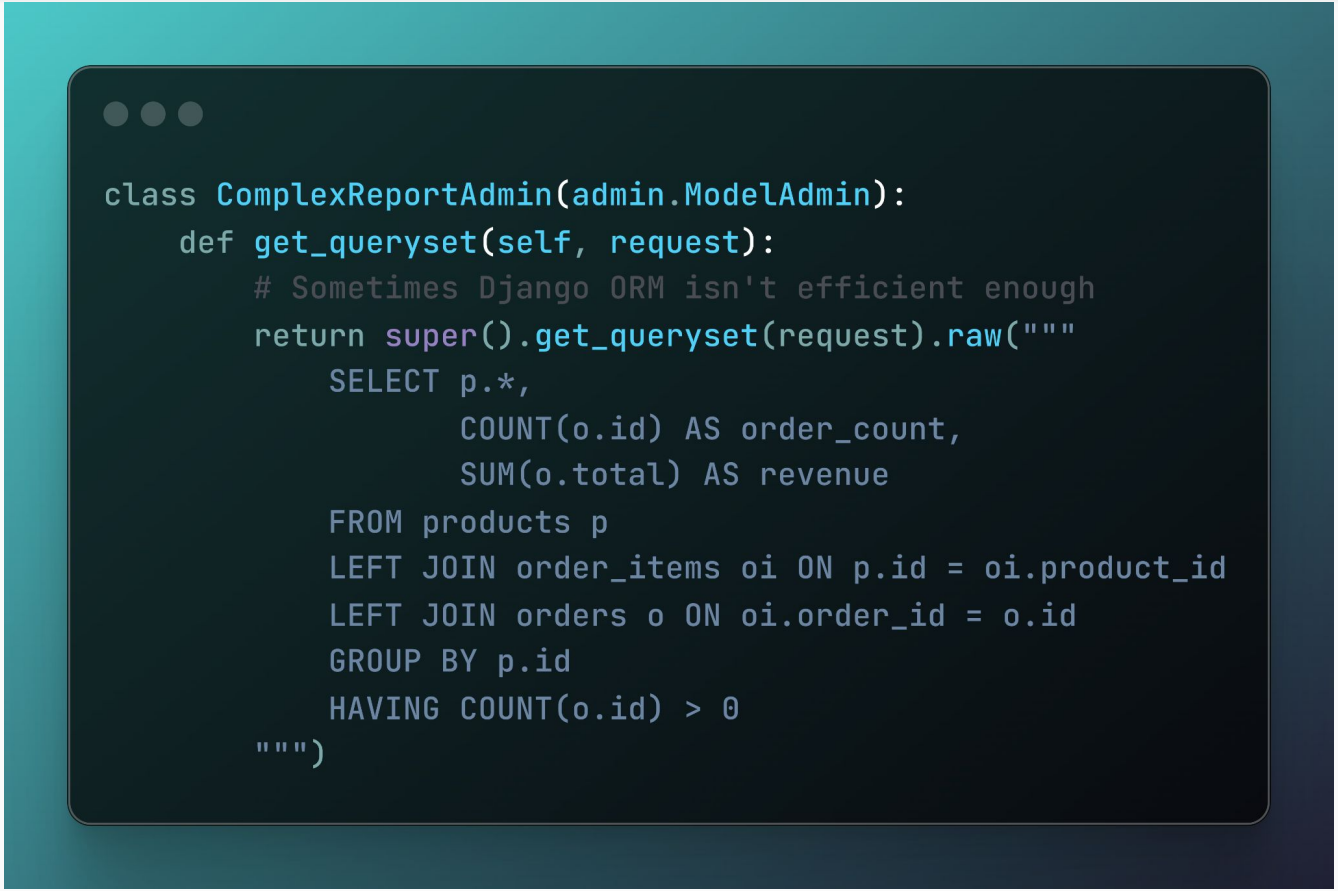
```
# Before: Inefficient
class StandardCategoryFilter(admin.SimpleListFilter):
    # Loads all categories for every request

# After: Optimized
class EfficientCategoryFilter(admin.SimpleListFilter):
    def lookups(self, request, model_admin):
        # Cache results
        cache_key = "category_filter_options"
        options = cache.get(cache_key)

        if not options:
            # Use values_list for efficiency
            options = list(
                Category.objects.values_list('id', 'name')
                .order_by('name')[:100]  # Limit options
            )
            cache.set(cache_key, options, 3600)

        return options
```

Solution #7: Raw SQL When Needed



```
class ComplexReportAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        # Sometimes Django ORM isn't efficient enough
        return super().get_queryset(request).raw("""
            SELECT p.*,
                   COUNT(o.id) AS order_count,
                   SUM(o.total) AS revenue
            FROM products p
            LEFT JOIN order_items oi ON p.id = oi.product_id
            LEFT JOIN orders o ON oi.order_id = o.id
            GROUP BY p.id
            HAVING COUNT(o.id) > 0
        """)
```

Part 3: Maintenance

Keeping Performance Gains Over Time

Monitoring Strategy

- **Add custom middleware to track admin performance**
- **Setup alerts for slow admin pages**
- **Regular Profiling sessions**
- **Database index usage monitoring**

```python
class AdminPerformanceMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        if request.path.startswith('/admin/'):
            start_time = time.time()
            response = self.get_response(request)
            duration = time.time() - start_time

            if duration > 0.5:  # Threshold in seconds
                logger.warning(f"Slow admin request: {request.path} ({duration:.2f}s)")

            return response
        return self.get_response(request)
```

Performance Testing

- **Add admin-specific performance tests**
- **Measure with realistic data volumes**
- **Simulate real user interactions, not just page loads**

Sample Performance Test

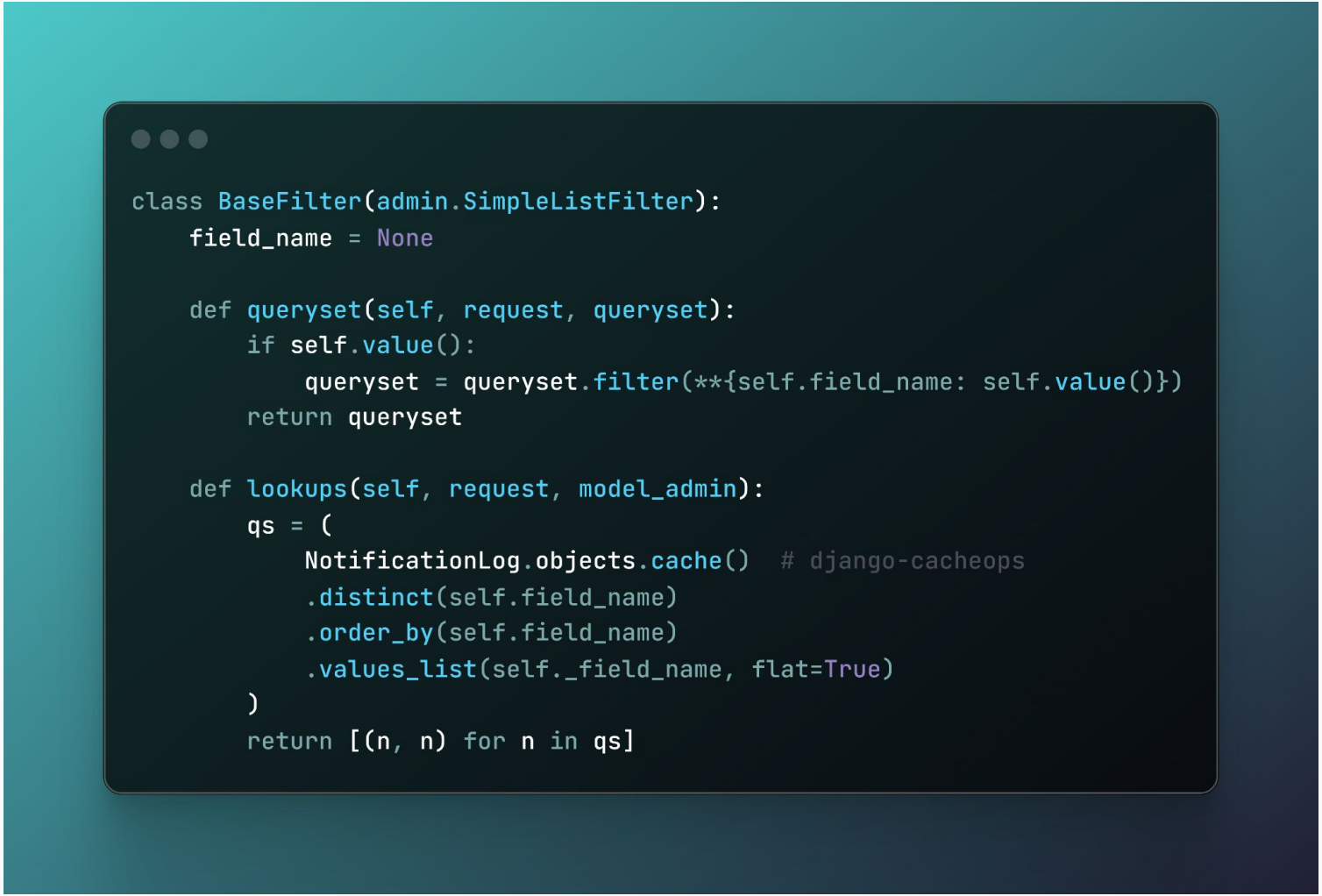
```python
class AdminPerformanceTest(TestCase):
    @classmethod
    def setUpClass(cls):
        # Generate 10,000 test records
        Product.objects.bulk_create(
            [
                Product(name=f"Product {i}", price=random.randint(10, 1000))
                for i in range(10000)
            ]
        )

    def test_product_list_load_time(self):
        start_time = time.time()
        response = self.client.get("/admin/store/product/")
        duration = time.time() - start_time

        self.assertTrue(duration < 0.5, f"Admin list page too slow:
{duration:.2f}s")
```

Real World Example

```python
@admin.register(NotificationLog)
class NotificationLogAdmin(admin.ModelAdmin):
    list_per_page = 50
    list_display = [
        "created_at",
        "email_address",
        "code",
        "category",
        "event",
        "template",
        "type",
    ]
    search_fields = ["=user_email", "=user__username", "=user__id"]
    list_filter = [
        EventFilter,
        CategoryFilter,
        CodeFilter,
        TemplateFilter,
        DateFilter,  # Returns Ranges for Dates
        TypeFilter,
    ]
    readonly_fields = ["user"]
    list_select_related = ["user"]
    paginator = NoCountPaginator  # dumb paginator
```


class BaseFilter(admin.SimpleListFilter):
field_name = None
def queryset(self, request, queryset):
if self.value():
queryset = queryset.filter(**{self.field_name: self.value()})
return queryset
def lookups(self, request, model_admin):
qs = (
NotificationLog.objects.cache() # django-cacheops
.distinct(self.field_name)
.order_by(self.field_name)
.values_list(self._field_name, flat=True)
)
return [(n, n) for n in qs]

How we refactored our filters

```python
from django.db import models


class Categories(models.TextChoices):
    ELECTRONICS = "EL", "Electronics"
    CLOTHING = "CL", "Clothing"
    SPORTS = "SP", "Sports"

    @classmethod
    def value_pairs(cls):
        return [(category, category) for category in cls.values()]


class CategoryFilter(BaseFilter):
    title = _("Category")
    parameter_name = "category"
    _field_name = "category"

    def lookups(self, request, model_admin):
        return Categories.value_pairs()
```

The Next thing...

```python
@admin.register(NotificationLog)
class NotificationLogAdmin(admin.ModelAdmin):
    ...

    # the next thing we did was started utilising
    # .only() and .defer()
    deferred_fields = ["field_a", "field_b", ...]

    def get_queryset(self, request):
        """
        Utilising django-cacheops for queryset caching
        """
        qs = super().get_queryset(request)
        # if we have foreign key/reverse foreign key/m2m relations
        # we can also do something like this
        # qs = qs.prefetch_related(Prefetch("user", queryset=CustomUsersQS)
        # qs = qs.select_related("field_a", field_b")
        return qs.defer(*self.deferred_fields).cache(ops=[...], timeout=5 * 60)
```

Key Takeaways

- **Diagnose with real data and tools**
- **Think about query counts, not just speed**
- **Use Django's built-in tools first (select_related, only, defer, etc.)**
- **Cache strategically**
- **Monitor and maintain performance**

Thanks
For Listening

@sumit4613

https://sumit4613.github.io/