# /sentry-django-perf-review

**Source:** `~/.claude/skills/sentry-django-perf-review/SKILL.md`

---

name: django-perf-review description: Django performance code review. Use when asked to "review Django performance", "find N+1 queries", "optimize Django", "check queryset performance", "database performance", "Django ORM issues", or audit Django code for performance problems. allowed-tools: Read Grep Glob Bash Task license: LICENSE

# Django Performance Review

Review Django code for **validated** performance issues. Research the codebase to confirm issues before reporting. Report only what you can prove.

# Review Approach

1. **Research first** - Trace data flow, check for existing optimizations, verify data volume
2. **Validate before reporting** - Pattern matching is not validation
3. **Zero findings is acceptable** - Don't manufacture issues to appear thorough
4. **Severity must match impact** - If you catch yourself writing "minor" in a CRITICAL finding, it's not critical. Downgrade or skip it.

# Impact Categories

Issues are organized by impact. Focus on CRITICAL and HIGH - these cause real problems at scale.

| Priority | Category | Impact |
|---|---|---|
| 1 | N+1 Queries | **CRITICAL** - Multiplies with data, causes timeouts |
| 2 | Unbounded Querysets | **CRITICAL** - Memory exhaustion, OOM kills |
| 3 | Missing Indexes | **HIGH** - Full table scans on large tables |
| 4 | Write Loops | **HIGH** - Lock contention, slow requests |
| 5 | Inefficient Patterns | **LOW** - Rarely worth reporting |

---

# Priority 1: N+1 Queries (CRITICAL)

**Impact:** Each N+1 adds `O(n)` database round trips. 100 rows = 100 extra queries. 10,000 rows = timeout.

# Rule: Prefetch related data accessed in loops

Validate by tracing: View → Queryset → Template/Serializer → Loop access

```
# PROBLEM: N+1 - each iteration queries profile
def user_list(request):
    users = User.objects.all()
    return render(request, 'users.html', {'users': users})

# Template:
```

```
# {% for user in users %}
#     {{ user.profile.bio }}  ← triggers query per user
# {% endfor %}

# SOLUTION: Prefetch in view
def user_list(request):
    users = User.objects.select_related('profile')
    return render(request, 'users.html', {'users': users})
```

# Rule: Prefetch in serializers, not just views

DRF serializers accessing related fields cause N+1 if queryset isn't optimized.

```
# PROBLEM: SerializerMethodField queries per object
class UserSerializer(serializers.ModelSerializer):
    order_count = serializers.SerializerMethodField()

    def get_order_count(self, obj):
        return obj.orders.count()  # ← query per user

# SOLUTION: Annotate in viewset, access in serializer
class UserViewSet(viewsets.ModelViewSet):
    def get_queryset(self):
        return User.objects.annotate(order_count=Count('orders'))

class UserSerializer(serializers.ModelSerializer):
    order_count = serializers.IntegerField(read_only=True)
```

# Rule: Model properties that query are dangerous in loops

```
# PROBLEM: Property triggers query when accessed
class User(models.Model):
    @property
    def recent_orders(self):
        return self.orders.filter(created__gte=last_week)[:5]

# Used in template loop = N+1
```

```
# SOLUTION: Use Prefetch with custom queryset, or annotate
```

## Validation Checklist for N+1

- [ ] Traced data flow from view to template/serializer
- [ ] Confirmed related field is accessed inside a loop
- [ ] Searched codebase for existing select_related/prefetch_related
- [ ] Verified table has significant row count (1000+)
- [ ] Confirmed this is a hot path (not admin, not rare action)

---

# Priority 2: Unbounded Querysets (CRITICAL)

**Impact:** Loading entire tables exhausts memory. Large tables cause OOM kills and worker restarts.

## Rule: Always paginate list endpoints

```
# PROBLEM: No pagination - loads all rows
class UserListView(ListView):
    model = User
    template_name = 'users.html'

# SOLUTION: Add pagination
class UserListView(ListView):
    model = User
    template_name = 'users.html'
    paginate_by = 25
```

## Rule: Use iterator() for large batch processing

```
# PROBLEM: Loads all objects into memory at once
for user in User.objects.all():
    process(user)
```

```
# SOLUTION: Stream with iterator()
for user in User.objects.iterator(chunk_size=1000):
    process(user)
```

## Rule: Never call list() on unbounded querysets

```
# PROBLEM: Forces full evaluation into memory
all_users = list(User.objects.all())

# SOLUTION: Keep as queryset, slice if needed
users = User.objects.all()[:100]
```

## Validation Checklist for Unbounded Querysets

- ☐ Table is large (10k+ rows) or will grow unbounded
- ☐ No pagination class, paginate_by, or slicing
- ☐ This runs on user-facing request (not background job with chunking)

---

# Priority 3: Missing Indexes (HIGH)

**Impact:** Full table scans. Negligible on small tables, catastrophic on large ones.

## Rule: Index fields used in WHERE clauses on large tables

```
# PROBLEM: Filtering on unindexed field
# User.objects.filter(email=email)  # full scan if no index

class User(models.Model):
    email = models.EmailField()  # ← no db_index

# SOLUTION: Add index
class User(models.Model):
    email = models.EmailField(db_index=True)
```

# Rule: Index fields used in ORDER BY on large tables

```
# PROBLEM: Sorting requires full scan without index
Order.objects.order_by('-created')

# SOLUTION: Index the sort field
class Order(models.Model):
    created = models.DateTimeField(db_index=True)
```

# Rule: Use composite indexes for common query patterns

```
class Order(models.Model):
    user = models.ForeignKey(User)
    status = models.CharField(max_length=20)
    created = models.DateTimeField()

    class Meta:
        indexes = [
            models.Index(fields=['user', 'status']),  # for filter(user=x, status=y)
            models.Index(fields=['status', '-created']),  # for filter(status=x).order_by('-
created')
        ]
```

# Validation Checklist for Missing Indexes

- ☐ Table has 10k+ rows
- ☐ Field is used in filter() or order_by() on hot path
- ☐ Checked model - no db_index=True or Meta.indexes entry
- ☐ Not a foreign key (already indexed automatically)

---

# Priority 4: Write Loops (HIGH)

**Impact:** N database writes instead of 1. Lock contention. Slow requests.

# Rule: Use bulk_create instead of create() in loops

```
# PROBLEM: N inserts, N round trips
for item in items:
    Model.objects.create(name=item['name'])

# SOLUTION: Single bulk insert
Model.objects.bulk_create([
    Model(name=item['name']) for item in items
])
```

# Rule: Use update() or bulk_update instead of save() in loops

```
# PROBLEM: N updates
for obj in queryset:
    obj.status = 'done'
    obj.save()

# SOLUTION A: Single UPDATE statement (same value for all)
queryset.update(status='done')

# SOLUTION B: bulk_update (different values)
for obj in objects:
    obj.status = compute_status(obj)
Model.objects.bulk_update(objects, ['status'], batch_size=500)
```

# Rule: Use delete() on queryset, not in loops

```
# PROBLEM: N deletes
for obj in queryset:
    obj.delete()

# SOLUTION: Single DELETE
queryset.delete()
```

## Validation Checklist for Write Loops

- [ ] Loop iterates over 100+ items (or unbounded)
- [ ] Each iteration calls create(), save(), or delete()
- [ ] This runs on user-facing request (not one-time migration script)

---

# Priority 5: Inefficient Patterns (LOW)

**Rarely worth reporting.** Include only as minor notes if you're already reporting real issues.

## Pattern: count() vs exists()

```
# Slightly suboptimal
if queryset.count() > 0:
    do_thing()

# Marginally better
if queryset.exists():
    do_thing()
```

**Usually skip** - difference is <1ms in most cases.

## Pattern: len(queryset) vs count()

```
# Fetches all rows to count
if len(queryset) > 0:  # bad if queryset not yet evaluated

# Single COUNT query
if queryset.count() > 0:
```

**Only flag** if queryset is large and not already evaluated.

## Pattern: get() in small loops

```
# N queries, but if N is small (< 20), often fine
for id in ids:
    obj = Model.objects.get(id=id)
```

**Only flag** if loop is large or this is in a very hot path.

---

# Validation Requirements

Before reporting ANY issue:

1. **Trace the data flow** - Follow queryset from creation to consumption
2. **Search for existing optimizations** - Grep for select_related, prefetch_related, pagination
3. **Verify data volume** - Check if table is actually large
4. **Confirm hot path** - Trace call sites, verify this runs frequently
5. **Rule out mitigations** - Check for caching, rate limiting

**If you cannot validate all steps, do not report.**

---

# Output Format

```
## Django Performance Review: [File/Component Name]

### Summary
Validated issues: X (Y Critical, Z High)

### Findings

#### [PERF-001] N+1 Query in UserListView (CRITICAL)
**Location:** `views.py:45`

**Issue:** Related field `profile` accessed in template loop without prefetch.

**Validation:**
- Traced: UserListView → users queryset → user_list.html → `{{ user.profile.bio }}` in loop
- Searched codebase: no select_related('profile') found
- User table: 50k+ rows (verified in admin)
- Hot path: linked from homepage navigation

**Evidence:**
```python
```
```

```
def get_queryset(self):
    return User.objects.filter(active=True)  # no select_related
```

**Fix:**

```
def get_queryset(self):
    return User.objects.filter(active=True).select_related('profile')
```

````
If no issues found: "No performance issues identified after reviewing [files] and validating [what you checked]."

**Before submitting, sanity check each finding:**
- Does the severity match the actual impact? ("Minor inefficiency" ≠ CRITICAL)
- Is this a real performance issue or just a style preference?
- Would fixing this measurably improve performance?

If the answer to any is "no" - remove the finding.

---

## What NOT to Report

- Test files
- Admin-only views
- Management commands
- Migration files
- One-time scripts
- Code behind disabled feature flags
- Tables with <1000 rows that won't grow
- Patterns in cold paths (rarely executed code)
- Micro-optimizations (exists vs count, only/defer without evidence)

### False Positives to Avoid

**Queryset variable assignment is not an issue:**
```python
# This is FINE - no performance difference
projects_qs = Project.objects.filter(org=org)
projects = list(projects_qs)
````
````

```
# vs this - identical performance
projects = list(Project.objects.filter(org=org))
```

Querysets are lazy. Assigning to a variable doesn't execute anything.

**Single query patterns are not N+1:**

```
# This is ONE query, not N+1
projects = list(Project.objects.filter(org=org))
```

N+1 requires a loop that triggers additional queries. A single `list()` call is fine.

**Missing select_related on single object fetch is not N+1:**

```
# This is 2 queries, not N+1 - report as LOW at most
state = AutofixState.objects.filter(pr_id=pr_id).first()
project_id = state.request.project_id  # second query
```

N+1 requires a loop. A single object doing 2 queries instead of 1 can be reported as LOW if relevant, but never as CRITICAL/HIGH.

**Style preferences are not performance issues:** If your only suggestion is "combine these two lines" or "rename this variable" - that's style, not performance. Don't report it.

---

Revision #4
Created 2026-02-18 08:40:00 UTC by John
Updated 2026-05-31 20:01:33 UTC by John