Django API Optimization Guide

Performance Impact Summary

🚀 Immediate Impact (50-90% improvement)

- **Prefetching**: Reduces 100+ queries to 2-3 queries
- **Database Indexes**: 10-100x faster filtering and searching
- **Caching Fields**: Eliminates expensive COUNT() operations

📈 Scalability Impact

- **Soft Delete**: Maintains performance as data grows
- **Denormalization**: Dashboard queries stay fast regardless of historical data
- **Custom QuerySets**: Consistent optimization patterns across your API

🎯 User Experience Impact

- **Search Optimization**: Sub-second search results even with thousands of records
- **Pagination**: Large datasets load smoothly
- **Response Caching**: Frequently accessed data serves instantly

Implementation Priority

1. **Start with**: Database indexes and prefetching (biggest immediate impact)
2. **Then add**: Custom QuerySets and caching fields
3. **Advanced**: Full-text search and denormalization for complex analytics

---

4. Soft Delete Pattern

What it is:

Instead of permanently deleting records, mark them as deleted with a timestamp.

python

```python
class BaseModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    deleted_at = models.DateTimeField(null=True, blank=True)

    objects = models.Manager()  # All records
    active_objects = ActiveManager()  # Only non-deleted records

    class Meta:
        abstract = True

    def soft_delete(self):
        self.deleted_at = timezone.now()
        self.save()

    def is_deleted(self):
        return self.deleted_at is not None

class ActiveManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(deleted_at__isnull=True)
```

API Benefits:

- **Data Recovery**: Can restore "deleted" records
- **Audit Trail**: Keep history of what was deleted and when
- **Referential Integrity**: Related records don't break when something is "deleted"
- **Better Analytics**: Historical data remains for reporting

Usage in APIs:

python

```python
# Instead of hard delete
employee.delete()  # Permanent deletion

# Use soft delete
employee.soft_delete()  # Recoverable deletion

# Query active records
active_employees = Employee.active_objects.all()
```

---

5. Caching Fields for Performance

What it is:

Store computed values in the database to avoid expensive calculations on every API call.

```python
class Branch(models.Model):
    # ... existing fields ...
    employee_count = models.PositiveIntegerField(default=0)
    active_employee_count = models.PositiveIntegerField(default=0)
    last_meal_log_date = models.DateTimeField(null=True, blank=True)

    def update_counts(self):
        self.employee_count = self.branch_employees.count()
        self.active_employee_count = self.branch_employees.filter(is_active=True).count()
        self.save(update_fields=['employee_count', 'active_employee_count'])

class Company(models.Model):
    total_branches = models.PositiveIntegerField(default=0)
    total_employees = models.PositiveIntegerField(default=0)
    active_branches = models.PositiveIntegerField(default=0)
```

API Benefits:

- **Faster Responses**: No need to count records on every request
- **Reduced Database Load**: Fewer complex queries
- **Consistent Performance**: Response time doesn't increase with data growth

Update Strategies:

```python
# Using Django signals
from django.db.models.signals import post_save, post_delete

@receiver(post_save, sender=Employee)
def update_branch_counts(sender, instance, **kwargs):
    if instance.branch:
        instance.branch.update_counts()

# Or periodic tasks with Celery
@periodic_task(run_every=crontab(minute=0))  # Every hour
def update_all_counts():
    for branch in Branch.objects.all():
        branch.update_counts()
```

---

6. Custom QuerySets for Query Optimization

What it is:

Pre-defined query methods that include optimal database operations.

```python
class EmployeeQuerySet(models.QuerySet):
    def with_related(self):
        """Load related data in single query"""
        return self.select_related(
            'branch', 'department', 'employee_type', 'canteen'
        ).prefetch_related('employee_meal_logs')

    def active(self):
        return self.filter(is_active=True, deleted_at__isnull=True)

    def by_branch(self, branch_id):
        return self.filter(branch_id=branch_id)

    def with_meal_summary(self):
        """Add meal statistics"""
        return self.annotate(
            total_meals=Count('employee_meal_logs'),
            meals_this_month=Count(
                'employee_meal_logs',
                filter=Q(employee_meal_logs__created_at__month=timezone.now().month)
            )
        )

    def search(self, query):
        """Full-text search across multiple fields"""
        return self.filter(
            Q(name__icontains=query) |
            Q(employee_id__icontains=query) |
            Q(branch__name__icontains=query)
        )

class Employee(models.Model):
    # ... fields ...
    objects = EmployeeQuerySet.as_manager()
```

API Benefits:

- **Consistent Queries**: Same optimization patterns across your app
- **Reduced N+1 Problems**: Built-in related data loading
- **Maintainable Code**: Complex queries defined once, used everywhere

Usage in API Views:

python

```python
# Bad: Multiple database hits
def get_employees(request):
    employees = Employee.objects.all()
    # Each employee access triggers additional queries for branch, department, etc.

# Good: Single optimized query
def get_employees(request):
    employees = Employee.objects.with_related().active()
    # All data loaded in one query
```

---

9. Prefetch Strategies

What it is:

Loading related data efficiently to avoid N+1 query problems.

python

```python
# Problem: N+1 queries
employees = Employee.objects.all()
for employee in employees:  # 1 query
    print(employee.branch.name)  # N additional queries

# Solution: Prefetching
employees = Employee.objects.select_related('branch', 'department')
for employee in employees:  # 1 query total
    print(employee.branch.name)  # No additional queries
```

Types of Prefetching:

select_related (for ForeignKey and OneToOne):

python

```python
# Loads related data in same query using JOINs
employees = Employee.objects.select_related(
    'branch',
    'department',
    'canteen',
    'branch__company'  # Can chain relationships
)
```

prefetch_related (for ManyToMany and reverse ForeignKey):

```python
# Loads related data in separate optimized queries
branches = Branch.objects.prefetch_related(
    'branch_employees',  # All employees for each branch
    'branch_departments',  # All departments for each branch
    'branch_employees__employee_meal_logs'  # Meal logs for all employees
)
```

Advanced Prefetching:

```python
from django.db.models import Prefetch

# Custom prefetch with filtering
recent_logs = MealLog.objects.filter(
    created_at__gte=timezone.now() - timedelta(days=30)
)

employees = Employee.objects.prefetch_related(
    Prefetch('employee_meal_logs', queryset=recent_logs, to_attr='recent_meals')
)

# Now employee.recent_meals contains only last 30 days
```

API Performance Impact:

```python
# Before optimization: 1 + N queries
# GET /api/employees/ with 100 employees = 101 database queries

# After optimization: 2-3 queries total
employees = Employee.objects.select_related('branch', 'department').prefetch_related('employee_meal_logs')
```

---

11. Search Fields and Full-Text Search

What it is:

Optimized search functionality for API endpoints.

Basic Search Implementation:

```python
class EmployeeQuerySet(models.QuerySet):
    def search(self, query):
        if not query:
            return self

        return self.filter(
            Q(name__icontains=query) |
            Q(employee_id__icontains=query) |
            Q(branch__name__icontains=query) |
            Q(department__name__icontains=query)
        )
```

PostgreSQL Full-Text Search:

```python
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank
from django.contrib.postgres.indexes import GinIndex

class Employee(models.Model):
    # ... existing fields ...
    search_vector = SearchVectorField(null=True)

    class Meta:
        indexes = [
            GinIndex(fields=['search_vector']),
        ]

# Update search vector
Employee.objects.update(
    search_vector=SearchVector('name', 'employee_id', 'branch__name')
)

# Search usage
def search_employees(query):
    search_query = SearchQuery(query)
    return Employee.objects.annotate(
        rank=SearchRank('search_vector', search_query)
    ).filter(search_vector=search_query).order_by('-rank')
```

API Benefits:

- **Fast Search**: Indexed search is much faster than LIKE queries
- **Relevance Ranking**: Results ordered by relevance
- **Multi-field Search**: Search across multiple fields simultaneously

---

12. Denormalization for Frequent Queries

What it is:

Storing computed/aggregated data in separate tables for faster access.

```python
class EmployeeDashboard(models.Model):
    """Denormalized data for dashboard APIs"""
    employee = models.OneToOneField(Employee, on_delete=models.CASCADE)

    # Cached company/branch info
    company_name = models.CharField(max_length=100)
    branch_name = models.CharField(max_length=100)
    department_name = models.CharField(max_length=100)

    # Meal statistics
    total_meals_current_month = models.PositiveIntegerField(default=0)
    total_meals_last_month = models.PositiveIntegerField(default=0)
    favorite_meal_type = models.CharField(max_length=20, null=True)
    last_meal_date = models.DateTimeField(null=True)

    # Cost information
    total_meal_cost_current_month = models.DecimalField(max_digits=10, decimal_places=2, default=0)

    updated_at = models.DateTimeField(auto_now=True)

class CompanyStats(models.Model):
    """Company-level statistics"""
    company = models.OneToOneField(Company, on_delete=models.CASCADE)
    total_employees = models.PositiveIntegerField(default=0)
    active_employees = models.PositiveIntegerField(default=0)
    total_branches = models.PositiveIntegerField(default=0)
    meals_served_today = models.PositiveIntegerField(default=0)
    meals_served_this_month = models.PositiveIntegerField(default=0)
    top_meal_type = models.CharField(max_length=20, null=True)
    updated_at = models.DateTimeField(auto_now=True)
```

Update Strategy:

```python
# Real-time updates via signals
@receiver(post_save, sender=MealLog)
def update_employee_dashboard(sender, instance, created, **kwargs):
  if created:
    dashboard, _ = EmployeeDashboard.objects.get_or_create(
      employee=instance.employee
    )
    dashboard.refresh_stats()

# Periodic batch updates
@periodic_task(run_every=crontab(minute=0))  # Every hour
def refresh_dashboard_stats():
  for dashboard in EmployeeDashboard.objects.all():
    dashboard.refresh_stats()
```

API Benefits:

- **Lightning Fast Dashboards**: Pre-computed data loads instantly
- **Complex Analytics**: Expensive calculations done offline
- **Consistent Performance**: Response time independent of historical data size

---

Additional Performance Recommendations

Database Connection Optimization:

```python
# settings.py
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'CONN_MAX_AGE': 600,  # Persistent connections
    'OPTIONS': {
      'MAX_CONNS': 20,  # Connection pooling
    }
  }
}
```

Query Debugging:

```python
# settings.py (development)
LOGGING = {
    'loggers': {
        'django.db.backends': {
            'level': 'DEBUG',  # Log all SQL queries
        }
    }
}

# Use django-debug-toolbar
INSTALLED_APPS = [
    'debug_toolbar',
]
```

API Response Caching:

```python
from django.views.decorators.cache import cache_page
from django.core.cache import cache

@cache_page(60 * 15)  # Cache for 15 minutes
def employee_list(request):
    return JsonResponse(employee_data)

# Or manual caching
def get_company_stats(company_id):
    cache_key = f'company_stats_{company_id}'
    stats = cache.get(cache_key)

    if stats is None:
        stats = calculate_company_stats(company_id)
        cache.set(cache_key, stats, 60 * 60)  # Cache 1 hour

    return stats
```

Pagination for Large Datasets:

```python
from django.core.paginator import Paginator
from rest_framework.pagination import PageNumberPagination

class StandardResultsSetPagination(PageNumberPagination):
    page_size = 50
    page_size_query_param = 'page_size'
    max_page_size = 1000

# Cursor pagination for very large datasets
from rest_framework.pagination import CursorPagination

class EmployeeCursorPagination(CursorPagination):
    page_size = 50
    ordering = 'id'  # Must have consistent ordering
```

These optimizations can dramatically improve API performance, reducing response times from seconds to milliseconds and supporting much larger datasets efficiently.