# Django Performance & You

Presented by @st3v3nmw (~~Software Engineer~~ Senior Intern)

# Agenda

0. Quick walkthrough on the Demo Project & Django Silk
1. The N+1 Query Problem
2. Interpreting EXPLAIN ANALYZE
3. Scan Types
4. Covering & Partial Indexes
5. Periodic Tasks

Link to code repository.

# Profile Before Optimizing

> Premature optimization is the root of all evil ~ Donald Knuth

The problem with premature optimization is that you never know in advance where the bottlenecks will be while coding the system.

A heuristic / rule of thumb to address this would be:

> 1. Make it work
> 2. Make it right
> 3. Use the system and find performance bottlenecks
> 4. Use a profiler in those bottlenecks to determine what needs to be optimized
> 5. Make it fast
>
> https://wiki.c2.com/?ProfileBeforeOptimizing=

### Performance Targets

*Ideally*, backend APIs should target a `p50` latency of `100 - 150ms`, and a `p95` latency of `250 - 300ms`. Some systems which will remain unnamed have tail latencies (`p99s, p99.9s`) greater than `20,000ms` 💀. Tail latencies affect your most valuable customers since they have more data, etc.

# Django Silk Setup

In `settings.py`, add the following:

```
INSTALLED_APPS = (
    ...
    "silk"
)

MIDDLEWARE = [
    "silk.middleware.SilkyMiddleware",
    ...
]
SILKY_PYTHON_PROFILER = True
SILKY_ANALYZE_QUERIES = True
SILKY_PYTHON_PROFILER_BINARY = True
SILKY_PYTHON_PROFILER_RESULT_PATH = "profiling/"
SILKY_EXPLAIN_FLAGS = {"costs": True, "verbose": True}
```

To enable access to the user interface add the following to your `urls.py`:

```
urlpatterns += [("silk/", include("silk.urls", namespace="silk"))]
```

Then run `migrate` & `collectstatic`.

# Example 1: The N+1 Query Problem

http://127.0.0.1:8000/api/messages/list_some/

*2,794ms overall* *484ms on queries* *103 queries*

```python
class MessageViewSet(ReadOnlyModelViewSet):
    """Message viewset."""

    @action(methods=["GET"], detail=False)
    def list_some(self, request):
        """List some messages."""
        messages = models.Message.objects.all()[:100]
        data = serializers.MessageSerializer(messages, many=True).data
        return Response(data=data, status=HTTP_200_OK)
```

```python
    @action(methods=["GET"], detail=False)
    def list_some(self, request):
        """List some messages."""
        messages = models.Message.objects.all()[:100]
        data = []
        for message in messages:
          sent_by = message.sent_by
          data.append({"sender": sent_by.full_name, "text": message.text})
        return Response(data=data, status=HTTP_200_OK)
```

# Example 1: Queries

*From http://127.0.0.1:8000/silk/requests/*

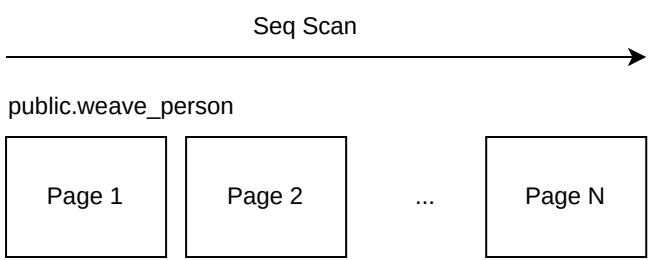
```
# Executed 1X -> Pick 100 messages
SELECT * FROM "weave_message" ORDER BY "weave_message"."updated" DESC LIMIT 100

# Executed 100X -> Once for each message
SELECT * FROM "weave_person" ORDER BY "weave_person"."updated" DESC
```

# Query Plan 1

SELECT * FROM "weave_person" ORDER BY "weave_person"."updated" DESC

*From http://127.0.0.1:8000/silk/request/GUID/sql/ID/*

```
Sort  (cost=73.83..76.33 rows=1000 width=76) (actual time=0.600..0.700 rows=1000 loops=1)
  Output: id, created, updated, deleted, first_name, last_name, email_address
  Sort Key: weave_person.updated DESC
  Sort Method: quicksort  Memory: 141kB
  ->  Seq Scan on public.weave_person  (cost=0.00..24.00 rows=1000 width=76) (actual time=0.013..0.207 rows=1000 loop
        Output: id, created, updated, deleted, first_name, last_name, email_address
Planning Time: 0.081 ms
Execution Time: 0.815 ms
```



# Example 1: Query Plan 2

*From http://127.0.0.1:8000/silk/request/GUID/sql/ID/*

SELECT * FROM "weave_message" ORDER BY "weave_message"."updated" DESC LIMIT 100

```
Limit  (cost=5515.46..5526.96 rows=100 width=104) (actual time=27.490..30.617 rows=100 loops=1)
  Output: id, created, updated, deleted, text, sent_by_id, thread_id
  ->  Gather Merge  (cost=5515.46..12280.22 rows=58824 width=104) (actual time=27.489..30.607 rows=100 loops=1)
        Output: id, created, updated, deleted, text, sent_by_id, thread_id
        Workers Planned: 1
        Workers Launched: 1
        ->  Sort  (cost=4515.45..4662.51 rows=58824 width=104) (actual time=24.413..24.418 rows=70 loops=2)
              Output: id, created, updated, deleted, text, sent_by_id, thread_id
              Sort Key: weave_message.updated DESC
              Sort Method: top-N heapsort  Memory: 61kB
              Worker 0:  actual time=21.745..21.752 rows=100 loops=1
                Sort Method: top-N heapsort  Memory: 62kB
              ->  Parallel Seq Scan on public.weave_message  (cost=0.00..2267.24 rows=58824 width=104) (actual time=0
                    Output: id, created, updated, deleted, text, sent_by_id, thread_id
                    Worker 0:  actual time=0.008..6.095 rows=43056 loops=1
Planning Time: 0.093 ms
Execution Time: 30.644 ms
```

# Example 1: Optimized

http://127.0.0.1:8000/api/messages/list_some/

*1,487ms overall* *82ms on queries* *1 queries*

```
1       @action(methods=["GET"], detail=False)
2       def list_some(self, request):
3           """List some messages."""
4           messages = (
5             models.Message.objects
6             .select_related("sent_by").all()[:100]
7           )
8           data = serializers.MessageSerializer(messages, many=True,).data
9           return Response(data=data, status=HTTP_200_OK)
```

## Queries

*From http://127.0.0.1:8000/silk/request/GUID/sql/ID/*

```
1   # Executed 1X -> Picks everything
2   SELECT * FROM "weave_message"
3   INNER JOIN "weave_person" ON ("weave_message"."sent_by_id" = "weave_person"."id")
4   ORDER BY "weave_message"."updated" DESC
5   LIMIT 100
```

# Example 1: Query Plan

```
Limit  (cost=5707.03..5718.53 rows=100 width=180) (actual time=77.875..81.329 rows=100 loops=1)
  Output: weave_message.id, weave_message.created, weave_message.updated, weave_message.deleted, weave_message.text, 
  ->  Gather Merge  (cost=5707.03..12471.79 rows=58824 width=180) (actual time=77.874..81.319 rows=100 loops=1)
        Output: weave_message.id, weave_message.created, weave_message.updated, weave_message.deleted, weave_message.
        ->  Sort  (cost=4707.02..4854.08 rows=58824 width=180) (actual time=73.949..73.957 rows=85 loops=2)
              Output: weave_message.id, weave_message.created, weave_message.updated, weave_message.deleted, weave_me
              Sort Key: weave_message.updated DESC
              Sort Method: top-N heapsort  Memory: 78kB
              ->  Hash Join  (cost=36.50..2458.80 rows=58824 width=180) (actual time=0.687..36.358 rows=50000 loops=2
                    Output: weave_message.id, weave_message.created, weave_message.updated, weave_message.deleted, we
                    Inner Unique: true
                    Hash Cond: (weave_message.sent_by_id = weave_person.id)
                    ->  Parallel Seq Scan on public.weave_message  (cost=0.00..2267.24 rows=58824 width=104) (actual 
                          Output: weave_message.id, weave_message.created, weave_message.updated, weave_message.delet
                    ->  Hash  (cost=24.00..24.00 rows=1000 width=76) (actual time=0.591..0.591 rows=1000 loops=2)
                          Output: weave_person.id, weave_person.created, weave_person.updated, weave_person.deleted, 
                          Buckets: 1024  Batches: 1  Memory Usage: 114kB
                          ->  Seq Scan on public.weave_person  (cost=0.00..24.00 rows=1000 width=76) (actual time=0.0
                                Output: weave_person.id, weave_person.created, weave_person.updated, weave_person.del
Planning Time: 0.459 ms
Execution Time: 81.380 ms
```

# Example 2: Unoptimized

http://127.0.0.1:8000/api/threads/

*113,450ms overall* *27,349ms on queries* *10,244 queries*

```
1   class ThreadViewSet(ReadOnlyModelViewSet):
2
3       queryset = models.Thread.objects.filter(deleted=False)
4       serializer_class = serializers.ThreadSerializer
5
6   class MessageSerializer(serializers.ModelSerializer):
7
8       sender = serializers.ReadOnlyField(source="sent_by.full_name")
9
10      class Meta:
11          model = Message
12          fields = "__all__"
13
14  class ThreadSerializer(serializers.ModelSerializer):
15
16      messages = MessageSerializer(many=True)
17
18      class Meta:
19          model = Thread
20          fields = "__all__"
```

# Example 2: Queries

```
# Executed 1X -> Picks 100 threads
SELECT * FROM "weave_thread"
WHERE NOT "weave_thread"."deleted"
ORDER BY "weave_thread"."updated" DESC
LIMIT 100

# Executed 100X -> Picks approx. 100 messages per thread
SELECT * FROM "weave_message"
WHERE "weave_message"."thread_id" = X
ORDER BY "weave_message"."updated" DESC

# Executed approx 10,000X -> Once for each message in each thread
SELECT * FROM "weave_person" WHERE "weave_person"."id" = X LIMIT 21

# Executed 1X -> Pagination
SELECT COUNT(*) AS "__count"
FROM "weave_thread"
WHERE NOT "weave_thread"."deleted"
# {
#    "count": 903,
#    "next": "http://127.0.0.1:8000/api/threads/?page=2",
#    "previous": null,
#    "results": [ ],
# }
```

# Example 2: After prefetching related messages

http://127.0.0.1:8000/api/threads/

*135,790ms overall* *34,587ms on queries* *10,145 queries*

```
class ThreadViewSet(ReadOnlyModelViewSet):
    queryset = (
      models.Thread.objects
      .filter(deleted=False)
      .prefetch_related("messages")
    )
```

## Queries

```
# Executed 1X -> Picks 100 threads
SELECT * FROM "weave_thread" WHERE NOT "weave_thread"."deleted" ORDER BY "weave_thread"."updated" DESC LIMIT 100

# Executed 1X -> Picks approx. 10,000 messages for all threads
SELECT * FROM "weave_message" WHERE "weave_message"."thread_id" IN (THREAD_IDS,)
ORDER BY "weave_message"."updated" DESC

# Executed approx 10,000X -> Once for each message in each thread
SELECT * FROM "weave_person" WHERE "weave_person"."id" = X LIMIT 21

SELECT COUNT(*) AS "__count" FROM "weave_thread" WHERE NOT "weave_thread"."deleted"
```

# Example 2: After prefetching related senders 😲

http://127.0.0.1:8000/api/threads/

*7,183ms overall* *1,360ms on queries* *4 queries*

```
class ThreadViewSet(ReadOnlyModelViewSet):
    queryset = (
      models.Thread.objects
      .filter(deleted=False)
      .prefetch_related("messages", "messages__sent_by")
    )
```

## Queries

```
SELECT * FROM "weave_thread" WHERE NOT "weave_thread"."deleted" ORDER BY "weave_thread"."updated" DESC LIMIT 100

# Executed 1X -> Picks approx. 10,000 messages for all threads
SELECT * FROM "weave_message" WHERE "weave_message"."thread_id" IN (THREAD_IDS,)
ORDER BY "weave_message"."updated" DESC

# Executed approx 1X -> Picks all relevant message senders
SELECT * FROM "weave_person" WHERE "weave_person"."id" IN (MESSAGE_IDS,)
ORDER BY "weave_person"."updated" DESC

SELECT COUNT(*) AS "__count" FROM "weave_thread" WHERE NOT "weave_thread"."deleted"
```

# Example 3: Covering Indexes

http://127.0.0.1:8000/api/messages/

88.364ms

```
# We will be focusing on only one query this time
# This query is run during pagination to get the total
# number of objects in the database

SELECT COUNT(*) AS "__count" FROM "weave_message"
WHERE NOT "weave_message"."deleted"
```

## Query Plan

```
Aggregate  (cost=2904.02..2904.03 rows=1 width=8) (actual time=23.799..23.800 rows=1 loops=1)
  Output: count(*)
  ->  Seq Scan on public.weave_message  (cost=0.00..2679.00 rows=90007 width=0) (actual time=0.008..17.689 rows=89964
        Output: id, created, updated, deleted, text, sent_by_id, thread_id
        Filter: (NOT weave_message.deleted)
        Rows Removed by Filter: 10036
Planning Time: 0.063 ms
Execution Time: 23.826 ms
```

# Example 3: Covering Indexes

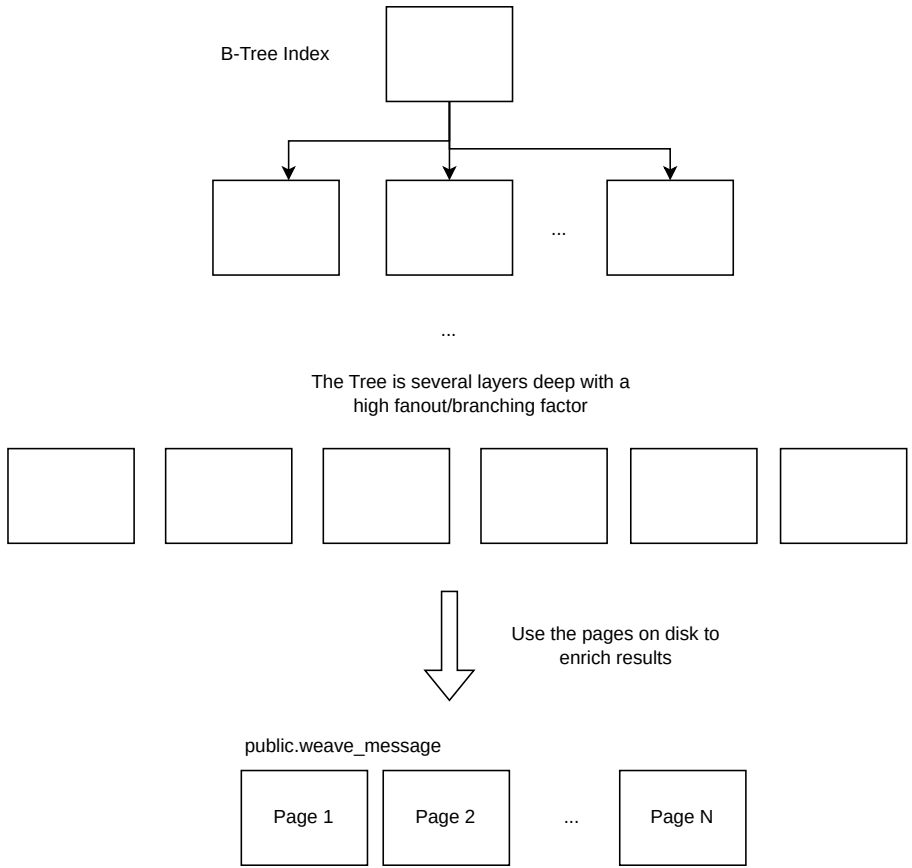
```python
class AbstractBase(models.Model):
    """Abstract Base."""

    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    deleted = models.BooleanField(db_index=True)

    class Meta:
        ordering = ("-updated",)
        abstract = True
```

## Query Plan

*14.985ms*

```
Aggregate  (cost=2116.43..2116.44 rows=1 width=8) (actual time=17.813..17.814 rows=1 loops=1)
  Output: count(*)
  ->  Index Only Scan using weave_message_deleted_9517d20b on public.weave_message  (cost=0.29..1891.41 rows=90007 wi
        Output: deleted
        Index Cond: (weave_message.deleted = false)
        Heap Fetches: 0
Planning Time: 0.078 ms
Execution Time: 17.846 ms
```

# Example 3: Covering Indexes



# Example 3: Partial Indexes

```python
from django.db.models import Index, Q

class Message(AbstractBase):
    class Meta(AbstractBase.Meta):
        indexes = [
            Index(
                fields=["deleted"],
                name="soft_deletes_ignore",
                condition=Q(deleted=False),
            )
        ]
```

## Query Plan

*21.83ms*

```
Aggregate  (cost=1887.20..1887.21 rows=1 width=8) (actual time=25.893..25.895 rows=1 loops=1)
  Output: count(*)
  ->  Index Only Scan using soft_deletes_ignore on public.weave_message  (cost=0.29..1662.18 rows=90007 width=0) (act
        Output: deleted
        Heap Fetches: 0
Planning Time: 0.137 ms
Execution Time: 25.931 ms
```

# Example 4: Index Scans

```
> from threads.weave.models import *
> deleted_messages = Message.objects.filter(deleted=True)
> str(deleted_messages.query)
SELECT * FROM "weave_message" WHERE "weave_message"."deleted" ORDER BY "weave_message"."updated" DESC
> deleted_messages.explain(analyze=True)  # Before Index
Sort  (cost=3342.87..3367.85 rows=9993 width=104) (actual time=21.625..22.368 rows=10036 loops=1)
  Sort Key: updated DESC
  Sort Method: quicksort  Memory: 1835kB
  ->  Seq Scan on weave_message  (cost=0.00..2679.00 rows=9993 width=104) (actual time=0.020..17.314 rows=10036 loops
        Filter: deleted
        Rows Removed by Filter: 89964
Planning Time: 0.099 ms
Execution Time: 23.005 ms
> deleted_messages.explain(analyze=True)  # After Index
Sort  (cost=2556.54..2581.52 rows=9993 width=104) (actual time=16.202..17.342 rows=10036 loops=1)
  Sort Key: updated DESC
  Sort Method: quicksort  Memory: 1835kB
  ->  Bitmap Heap Scan on weave_message  (cost=113.74..1892.67 rows=9993 width=104) (actual time=1.528..9.371 rows=10
        Filter: deleted
        Heap Blocks: exact=1676
        ->  Bitmap Index Scan on weave_message_deleted_9517d20b  (cost=0.00..111.24 rows=9993 width=0) (actual time=0
              Index Cond: (deleted = true)
Planning Time: 0.207 ms
Execution Time: 18.260 ms
```

# SELECT COUNT(*) Estimates

Do we really need exact total counts during pagination? 🤔

> Google Search Doesn't...
>
> > "About 25,200,000,000 results (0.45 seconds)"

```python
# threads.weave.paginators.py

class CustomPaginator(Paginator):

    @cached_property
    def count(self) -> int:
        """Return the total number of objects, across all pages."""
        with connection.cursor() as cursor:
            cursor.execute(
                "SELECT reltuples FROM pg_class WHERE relname = %s",
                [self.object_list.query.model._meta.db_table],
            )
            return int(cursor.fetchone()[0])

class CustomPageNumberPagination(PageNumberPagination):

    django_paginator_class = CustomPaginator
```

# SELECT COUNT(*) Estimates

*1.39ms*

## Query

```
SELECT reltuples FROM pg_class WHERE relname = weave_message
# Output = 100,000; Correct = 89,964 😖
# Filters/conditions not applied
```

> The catalog `pg_class` catalogs tables and most everything else that has columns or is otherwise similar to a table. This includes indexes, views, materialized views, composite types, and TOAST tables; see `relkind`. When we mean all of these kinds of objects we speak of "relations".
> `reltuples` is the number of live rows in the table. This is only an estimate used by the planner. It's updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`.
> ~ https://www.postgresql.org/docs/current/catalog-pg-class.html

## Query Plan

```
Index Scan using pg_class_relname_nsp_index on pg_class
  (cost=0.28..8.29 rows=1 width=4)
  (actual time=0.043..0.046 rows=1 loops=1)
```

Are we able to surface the count estimates from the query planner? Probably not.

# SELECT COUNT(*) Estimates

Introducing `TABLESAMPLE SYSTEM/BERNOULLI (n)` 🥁

*11.50ms*

## But can we do better? Yes!

> The `TABLESAMPLE SYSTEM/BERNOULLI` method returns an approximate percentage of rows. It generates a random number for each physical storage page for the underlying relation. Based on this random number and the sampling percentage specified, it either includes or exclude the corresponding storage page. If that page is included, the whole page will be returned in the result set.
> ~ https://wiki.postgresql.org/wiki/TABLESAMPLE_Implementation

```
SELECT COUNT(*) AS "__count" FROM "weave_message" TABLESAMPLE BERNOULLI (10) REPEATABLE (42)
WHERE NOT "weave_message"."deleted"
# Average output from several runs = 90,122.5; Correct = 89,964
# Off by ONLY 0.18%
```

## Query Plan

```
Aggregate (cost=1801.50..1801.51 rows=1 width=8) (actual time=11.505..11.506 rows=1 loops=1)
# Original time taken: Around 25ms
```

# SELECT COUNT(*) Estimates

```python
class CustomPaginator(Paginator):
    @cached_property
    def count(self) -> int:
        """Return an estimate of the total number of objects.
        Could be made more robust by first running the normal COUNT(*)
        with a `SET LOCAL statement_timeout TO 50`
        and then reverting to this if that doesn't complete in time.
        """
        with connection.cursor() as cursor:
            db_table = self.object_list.model._meta.db_table
            query = (
                self.object_list.annotate(__count=Count("*"))
                .values("__count")
                .order_by()
                .query
            )
            query.group_by = None
            query_string = str(query)
            query_string = query_string.replace(
                f'FROM "{db_table}"',
                f'FROM "{db_table}" TABLESAMPLE BERNOULLI (10) REPEATABLE (42)',
            )
            cursor.execute(query_string)
            return 10 * int(cursor.fetchone()[0])
```

# Periodic Tasks

- Periodic tasks are one of the 4 horsemen of the apocalypse 😉. They should be avoided like the plague.
  - Instead, prefer event-driven tasks
    - These typa tasks run immediately after an event e.g. `` `instance.save` ``, `` `instance.create` ``, `` `instance.transition` ``, Django signals, etc, ...
    - You can use Celery to set-up automatic retries for certain errors
      - Retry with exponential backoff to reduce load
  - For reporting, separate your transactional (Postgres) and analytical (BigQuery) databases so that reports don't slow down your main database
- If you must:
  - Do things in bulk e.g. `` `bulk_create` ``, `` `bulk_update` ``
  - Use iterators when looping over a large number of objects (`` `qs.iterator()` ``)
    - Helps reduce memory usage
  - Add indexes to fields that are used to filter. This speeds up reads but slows down writes (there's no such thing as a free lunch)

# Periodic Tasks

```
1   q = Entry.objects.filter(headline__startswith="What")
2   q = q.filter(pub_date__lte=datetime.date.today())
3   q = q.exclude(body_text__icontains="food")
4   print(q)
```

- Realize that querysets are lazy, so avoid things like:
  - `len(qs)` -> `qs.count()`
  - `list(qs)` 😒
  - `bool(qs)` -> `qs.exists()`

> QuerySets are lazy – the act of creating a QuerySet doesn't involve any database activity. You can stack filters together all day long, and Django won't actually run the query until the QuerySet is evaluated
> https://docs.djangoproject.com/en/4.2/ref/models/querysets/#when-querysets-are-evaluated

# Thank You!

And if everything fails, try API caching lol 🫠