Django at Scale

Adam Johnson - me@adamj.eu

12th August 2014

History

- Worked at Memrise.com as lead developer - 1.3 million users, 20k uniques/day
- Now at YPlan - over 1 million downloads, N uniques/day
- Both powered by Django with API and Web parts

Scaling Django

- What can you do with your code, caching, and database?

FIG. 2.
PONY "MAGIC"

Wrap Django's classes

- Wrap around django where sensible, to implement scalability changes at the highest level.
- Views, admin, models, querysets, fields, ...

```
# project/admin.py
class ModelAdmin(admin.ModelAdmin):
    pass
```

```
# blog/admin.py
from project.admin import ModelAdmin

class BlogPostAdmin(ModelAdmin):
    # bla bla bla...
```

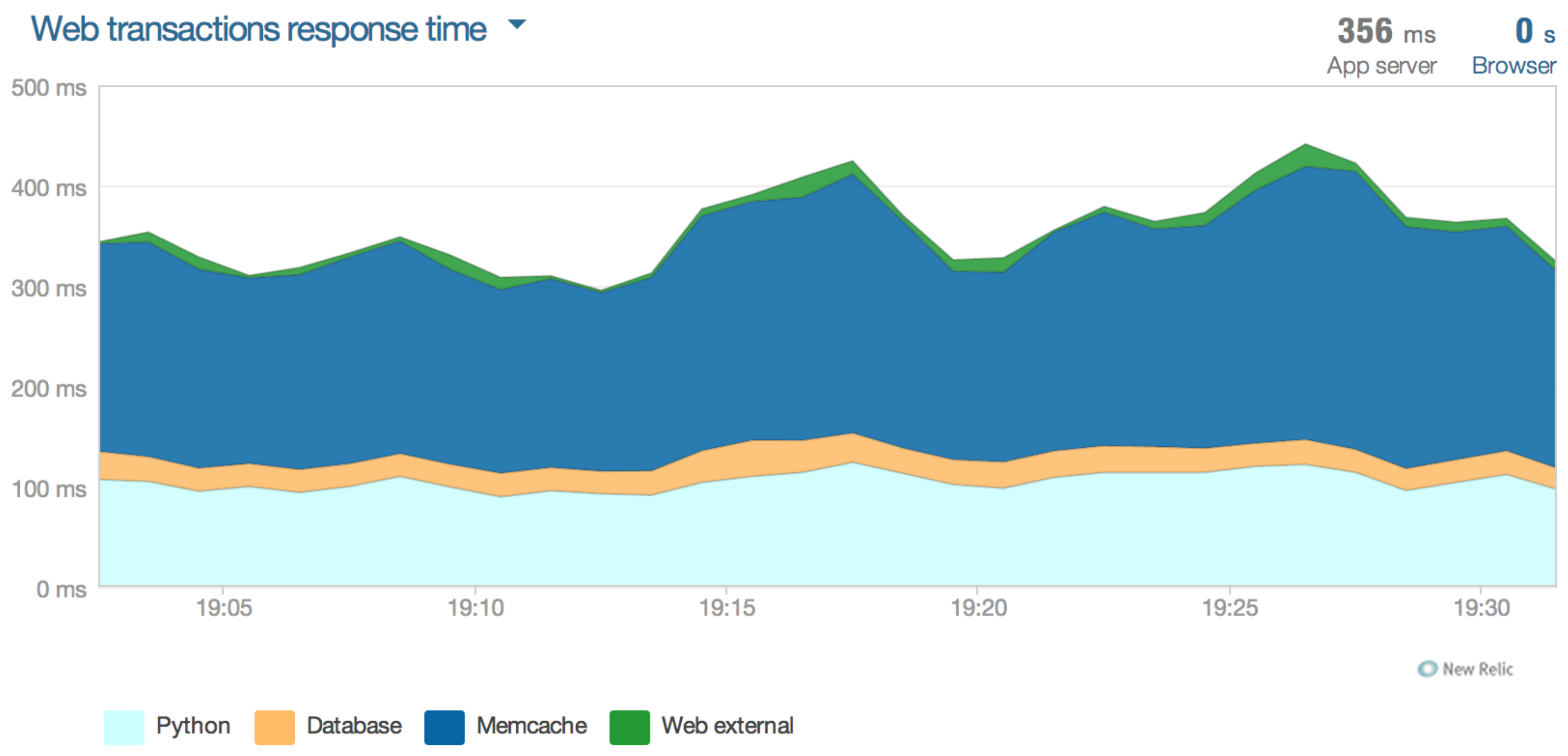
- e.g. to make all Admin pages use new queryset class (see my blog post on approximate counts)

```
# project/admin.py
class ModelAdmin(admin.ModelAdmin):
    def queryset(self, request):
        qs = super(ModelAdmin, self).queryset(request)
        qs = qs._clone(klass=ApproxCountQuerySet)
        return qs
```

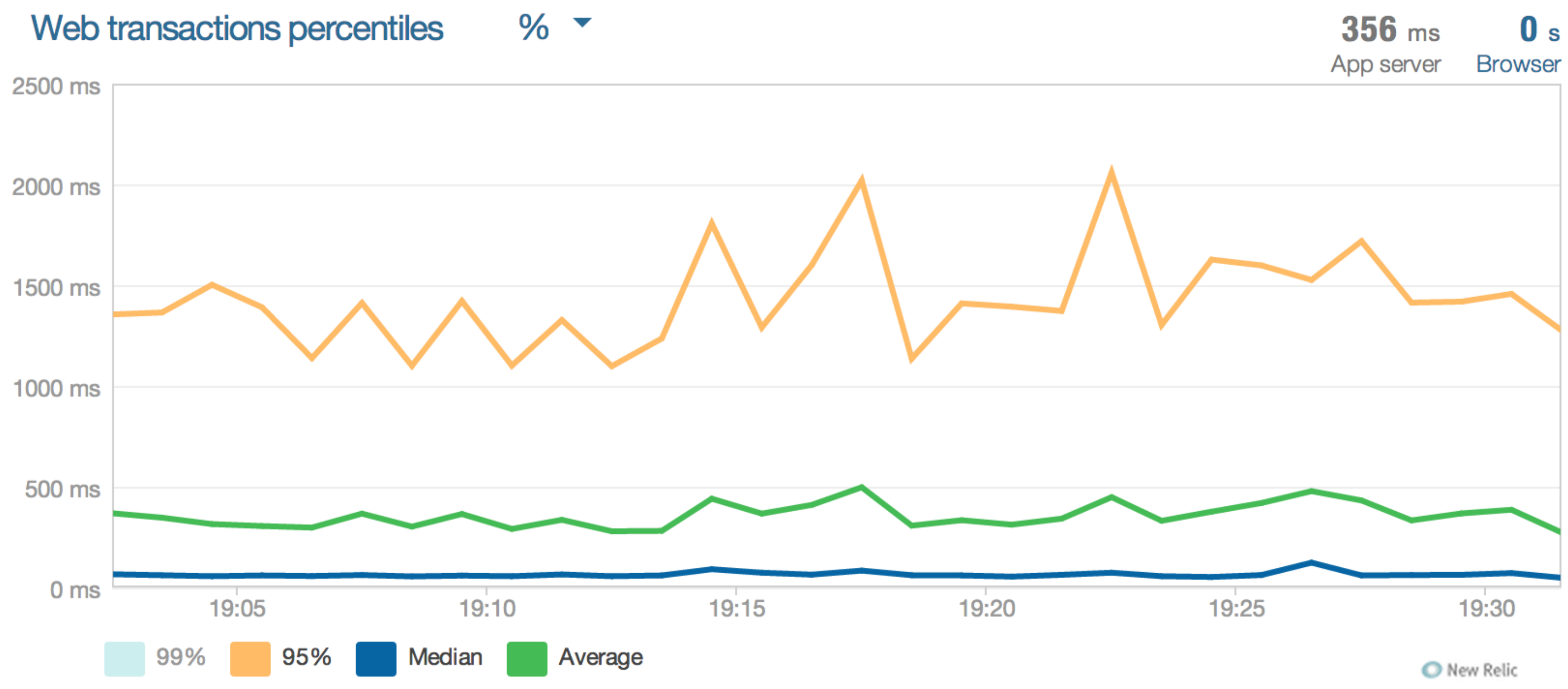
Caching

- Caches make things faster.
- But don't be overzealous - keep your caching/invalidation model simple, and don't use it to sticky tape over more fundamental problems (slow DB, code execution time, ...).

- Monitor all the things (New Relic):



- At scale, percentile view is much more useful:



Background fill your cache

- Typical cache code puts the refill in the request path:

```
def some_func(user_id):
    cache_key = 'key:' + str(user_id)
    val = cache.get(cache_key)
    if val is None:
        val = slow_func()
        cache.set(cache_key, val)
    return val
```

- For the 99th percentile power user, a reasonable but slightly slow function call could become way too slow (>100ms).

- Instead, use an always-there cache value with a timestamp:

```
def some_func():
    cache_key = 'key:' + str(user_id)
    last_update, val cache.get(cache_key)
    if last_update < now() - timedelta(minutes=15):
        # celery task to refill cache_key
        slow_func.apply_async([user_id])
    return val
```

- In-request code path now just one cache fetch for even the 99th percentile user on a bad day.

- (Code simplified - you want to put task in queue just once, and maybe use a persistent cache backend.)

Database

- Don't be hasty to get away from the ORM or relational databases.

- Instead, learn them better.

- Understand how Querysets become SQL, how the DB will handle SQL, and find easy wins on this path.

Use prefetch_related

- **select_related** on many models will generate a query with a lot of JOIN clauses.

- **prefetch_related** will generate many smaller simpler queries - one for each table.

- It may be counterintuitive, but using **prefetch_related** to do app-level JOINs can be faster and is much more scalable.

- Django is really friendly here - it's so easy to switch 'select' to 'prefetch'!

Learn the way of the index

- Indexing is an art that takes time to perfect. Some trial and error.

- Number one problem in Django - bunging 'index=True' on several fields and expecting all queries to magically speed up.

Run a query killer/sniper

- Web request timed out != query cancelled
- Every 10 seconds, kill any query lasting longer than 30 seconds, and email you about it

Use replicas smartly

- Replicas are amazing, use them.

- You can use a **DBRouter** in your Django settings to add logic for which DB is used.

- Or, you can manually direct certain queries that you know can handle slightly stale data, e.g.:

```
# project/admin.py
class ModelAdmin(admin.ModelAdmin):
    def queryset(self, request):
        qs = super(ModelAdmin, self).queryset(request)
        if request.method == 'GET':
            qs = qs.using('replica')
        return qs
```

- Routing admin to replica is actually a really big win - stop the big nasty admin queries from affecting the performance of the main app.

Thank you

- `me@adamj.eu`