Performance in Python Web frameworks

Django, Django REST, FastAPI

1. Common DRF usage
2. Utilities:
   - Django Debug Toolbar
   - Locust
3. Django / FastAPI *basic* features with QuerySet
4. QuerySet and Models relationships
5. QuerySet and ‘for-loop’
6. Django Async
7. Other techniques and suggestions

1.1 All tests will be with this DB model:

```
class Product(models.Model):
   low_price = models.DecimalField(max_digits=7, decimal_places=2)
   high_price = models.DecimalField(max_digits=7, decimal_places=2)

   demand_qty = models.IntegerField()
   offers_qty = models.IntegerField()

   bought_qty = models.IntegerField()
   sold_qty = models.IntegerField()

   time_created = models.DateTimeField()

   class Meta:
       ordering = ('time_created',)
```

Total: 10M objects in PostgreSQL

1.2 Simple DRF; target: Model **objects (instance)**

```
class ProductViewSet(ModelViewSet):
   permission_classes = (AllowAny,)
   serializer_class = ProductSerializer
   queryset = Product.objects.all()[:LIMIT_1K]


class ProductSerializer(serializers.ModelSerializer):
      class Meta:
             model = Product
             fields = (
                 'low_price',
                 'high_price',
                 'demand_qty',
                 'offers_qty',
                 'bought_qty',
                 'sold_qty',
                 'time_created',
             )
```

QS	Time, sec	Res.size, MB	RPS
1k	0.9	0.143	3
10k	1.38	1.4	-
100k	5.5	13.96	-
1M	~40	139.61 (3.1 Gb RAM)	-

1.3 Simple DRF; target: objects.**values()**

```
class ProductViewSet(ModelViewSet):
   permission_classes = (AllowAny,)
   serializer_class = ProductSerializer
   queryset = Product.objects.values()[:LIMIT_1K]


class ProductSerializer(serializers.ModelSerializer):
      class Meta:
             model = Product
             fields = (
                 'low_price',
                 'high_price',
                 'demand_qty',
                 'offers_qty',
                 'bought_qty',
                 'sold_qty',
                 'time_created',
             )
```

QS	Time, sec	Res.size, MB	RPS
1k	**1.06** **+17%**	0.143	-
10k	1.39	1.4	-
100k	**4.95** **-10%**	13.96	-
1M	**~35** **-12.5%**	139.61	-

1.4 **No** Serializer; target: objects**.values()**

```
class ProductsValues(APIView):
   permission_classes = (AllowAny,)

   @staticmethod
   def get(request, *args, **kwargs):
       return Response(
       data=Product.objects.values()[:LIMIT_1K]
       )
```

QS	Time, sec	Res.size, MB	RPS
1k	**0.57** **-36.6%**	0.148	**4.8** **+37.5%**
10k	0.66	1.45	-
100k	1.8	14.6	-
1M	**9.8** **-75.5%**	146.94	-

Standard DRF with Serializer is too slow!

1.5 Standard FaspAPI implementation; target: Model **object**

```
Model -> Query -> pydantic schema -> endpoint
```

QS	Time, sec	Res.size, MB	RPS
1k	0.065	152.6 KB	**43 x14**
10k	0.630	1.51	-
100k	**6.2 +12%**	15.18	-
1M	**n/a**	**n/a**	-

*compare to DRF in 1.2

FastAPi is fast.

1.6 **No** Serializer, ‘for-loop’; target: Model **objects**

```
class ForLoopObjects(APIView):
   permission_classes = (AllowAny,)

   @staticmethod
   def get(request, *args, **kwargs):
       out_data = {
           'low_price': [], 'high_price': [],
           'demand_qty': [], 'offers_qty': [],
           'bought_qty': [], 'sold_qty': [],
           'time_created': [],
       }
       for item in Product.objects.all()[:LIMIT 1K]:
            out_data['low_price'].append(item.low_price)
            out_data['high_price'].append(item.high_price)
            out_data['demand_qty'].append(item.demand_qty)
            out_data['offers_qty'].append(item.offers_qty)
            out_data['bought_qty'].append(item.bought_qty)
            out_data['sold_qty'].append(item.sold_qty)
            out_data['time_created'].append(item.time_created)
       return Response(data=out_data)
```

QS	Time, sec	Res.size, MB	RPS
1k	0.57	49.42 KB	-
10k	0.69	490.53 KB	-
100k	2.25	4.79	-
1M	14.5	47.86	-

-

1.7 **No** Serializer, ‘for-loop’; target: objects**.values()**

```
class ForLoopObjects(APIView):
   permission_classes = (AllowAny,)

   @staticmethod
   def get(request, *args, **kwargs):
       out_data = {
           'low_price': [], 'high_price': [],
           'demand_qty': [], 'offers_qty': [],
           'bought_qty': [], 'sold_qty': [],
           'time_created': [],
       }
       for item in Product.objects.values()[:LIMIT_1K]:
            out_data['low_price'].append(
               item.get('low_price'))
            out_data['high_price'].append(
               item.get('high_price'))
            ................

       return Response(data=out_data)
```

QS	Time, sec	Res.size, MB	RPS
1k	0.57	49.42 KB	-
10k	**0.65**	490.53 KB	-
100k	**1.78**	4.79	-
1M	**9.0** **-38%**	47.86	-

‘For-loop’ over List[dict] is faster than over objects!

2. How to measure performance?

- Postman - response time and size
- Django debug toolbar - SQL queries view
- Locust - Requests per second (RPS)
- django-silk ?

2.1 Django debug toolbar

SQL queries from 1 connection

default 454.63 ms (101 queries including 100 similar)

Query	Timeline	Time (ms)	Action
SELECT ••• FROM "some_app_product" **ORDER BY** "some_app_product"."time_created" **ASC LIMIT** 100		403.26	Sel Expl
SELECT "some_app_product"."id", "some_app_product"."high_price" **FROM** "some_app_product" **WHERE** "some_app_product"."id" = 1 **LIMIT** 21 **100 similar queries.**		0.70	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 2 **LIMIT** 21 **100 similar queries.**		0.51	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 3 **LIMIT** 21 **100 similar queries.**		0.48	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 4 **LIMIT** 21 **100 similar queries.**		0.41	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 5 **LIMIT** 21 **100 similar queries.**		0.40	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 6 **LIMIT** 21 **100 similar queries.**		0.54	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 7 **LIMIT** 21 **100 similar queries.**		0.64	Sel Expl
SELECT ••• FROM "some_app_product" **WHERE** "some_app_product"."id" = 8 **LIMIT** 21 **100 similar queries.**		0.42	Sel Expl

Connection: default

```
/home/valik/PycharmProjects/django_10M_QS/venv/lib/python3.9/site-packages/django/contrib/staticfiles/handlers.py in __call__(76)
    return self.application(environ, start_response)

/home/valik/PycharmProjects/django_10M_QS/venv/lib/python3.9/site-packages/django/views/decorators/csrf.py in wrapped_view(54)
    return view_func(*args, **kwargs)

/home/valik/PycharmProjects/django_10M_QS/venv/lib/python3.9/site-packages/django/views/generic/base.py in view(70)
    return self.dispatch(request, *args, **kwargs)

/home/valik/PycharmProjects/django_10M_QS/venv/lib/python3.9/site-packages/rest_framework/views.py in dispatch(506)
    response = handler(request, *args, **kwargs)

/home/valik/PycharmProjects/django_10M_QS/some_app/views.py in get(144)
    out_data['high_price'].append(product.high_price)
```

Hide »

- History — /sql_debug_v2/
- Versions — Django 3.2.13
- Time — CPU: 838.85ms (1240.35ms)
- Settings
- Headers
- Request — SQLDebugV2
- SQL — 101 queries in 454.63ms
- Static files — 10 files used
- Templates — rest_framework/api.html
- Cache — 0 calls in 0.00ms
- Signals — 29 receivers of 15 signals
- Logging — 0 messages
- *Intercept redirects*
- *Profiling*

2.2 Locust - load testing tool

3. Basic features with QuerySet

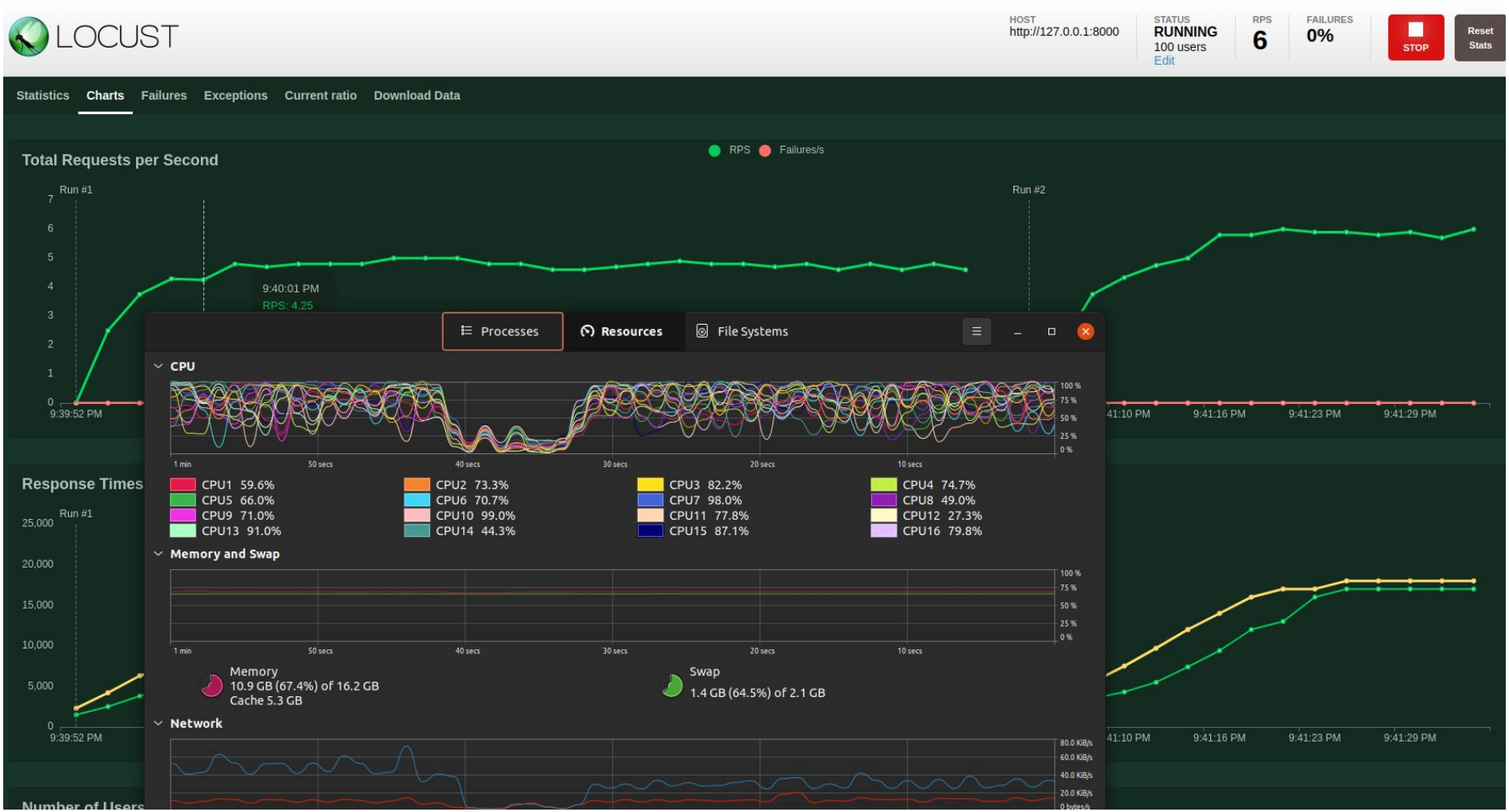
- Do you need model instance with ALL fields ?
- Do you need model instance or ‘object as dictionary’ will be enough?
- .only(), .defer(), .values(), .values_list() QuerySet methods.

3.1 Query ‘all’ vs ‘2’ fields

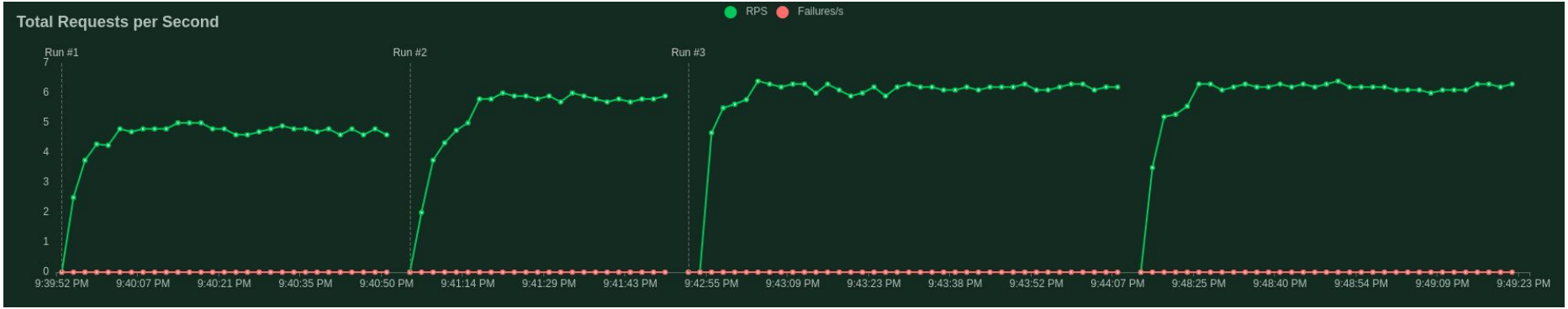
```
out_data = {'low_price': [], 'high_price': []}

""" v1:   4.8 RPS;  SQL: 1q ~550ms """
products = Product.objects.all()[:LIMIT_100]

""" v2.1: 5.8 RPS;  SQL: 1q ~435ms;  +17%; """
products = Product.objects.only('low_price', 'high_price')[:LIMIT_100]

""" v2.2: 5.5 RPS;  SQL: 101q ~450ms """
products = Product.objects.only('low_price')[:LIMIT_100]

""" v3:   6.3 RPS;  SQL: 1q ~405ms;  +8% (total +25%);   response: 440ms """
products = Product.objects.values('low_price', 'high_price')[:LIMIT_100]

""" FastAPI:  980 RPS;  response: 7ms """
products = db_session.query(Product).options(load_only(Product.low_price,Product.high_price))

for product in products:
   out_data['low_price'].append(product.low_price)     # product.get('low_price')
   out_data['high_price'].append(product.high_price)   # product.get('high_price')
```

3.2 Bonus - what will happened with 10M QuerySet ?

```
Django:

def get(request, *args, **kwargs):
   out_data = {'low_price': [], 'high_price': []}

   """ ? RPS;  SQL: ?q ?ms;  response: ? ms """
   products = Product.objects.all()

   return Response(data=out_data)


……................................

SQLAlchemy:

data = db_session.query(models.Product).offset(skip).limit(limit)  # .all()
```


LOCUST
HOST
http://127.0.0.1:8000
STATUS
RUNNING
100 users
Edit
RPS
6
FAILURES
0%
STOP
Reset Stats
Statistics
Charts
Failures
Exceptions
Current ratio
Download Data
Total Requests per Second
RPS
Failures/s
Run #1
Run #2
9:40:01 PM
RPS: 4.25
9:39:52 PM
9:41:10 PM
9:41:16 PM
9:41:23 PM
9:41:29 PM
Response Times
Number of Users
Processes
Resources
File Systems
CPU
CPU1 59.6%
CPU2 73.3%
CPU3 82.2%
CPU4 74.7%
CPU5 66.0%
CPU6 70.7%
CPU7 98.0%
CPU8 49.0%
CPU9 71.0%
CPU10 99.0%
CPU11 77.8%
CPU12 27.3%
CPU13 91.0%
CPU14 44.3%
CPU15 87.1%
CPU16 79.8%
Memory and Swap
Memory
10.9 GB (67.4%) of 16.2 GB
Cache 5.3 GB
Swap
1.4 GB (64.5%) of 2.1 GB
Network

Total RPS for 3.1



LOCUST

HOST
http://127.0.0.1:8000

STATUS
STOPPED
New test

RPS
993.2

FAILURES
0%

Statistics | Charts | Failures | Exceptions | Current ratio | Download Data

Total Requests per Second

RPS · Failures/s

1,200
1,000
800
600
400
200
0

1:58:55 PM · 1:58:57 PM · 1:58:59 PM · 1:59:01 PM · 1:59:03 PM · 1:59:05 PM · 1:59:07 PM · 1:59:09 PM · 1:59:11 PM · 1:59:13 PM · 1:59:15 PM · 1:59:17 PM · 1:59:20 PM · 1:59:22 PM

Response Times (ms)

Median Response Time · 95% percentile

120
100
80
60
40
20
0

1:58:55 PM · 1:58:57 PM · 1:58:59 PM · 1:59:0 · 59:20 PM · 1:59:22 PM

100% · Screenshot from 2022-04-29 13-59-08.png

Processes · Resources · File Systems

CPU

100 %
50 %

Number of Users

4. QuerySet and ORM relationships

QuerySet methods with ‘model objects’:

- .select_related(“FK1”, “O2O”, “FK2__FK3”)
- .prefetch_related(“M2M”)
- .only(“FK1__FK2__target_field”)

  Should be used for all model instances or only one!

Reach the edge of ORM with:

- .values(“FK1__FK2__FK3__target_field”)

4.1 select_related FK's == performance booster

```
return PlayerFightProgress.objects.filter(
    id__in=pfp_objs_ids,
).select_related(
    'character__profession',          # fk1 = character, fk2 = profession
    'fight__report',                  # fk3 = fight, fk4 = report
).order_by(
    '-damage_data__0__0__0',
).only(
    'id',
    'fight_id',
    'fight__report__is_cm',
    'character__profession__specialization',
    'character__profession__profession_icon',
).first()
```

Result: without .select_related() - 975ms, with - 600ms -> **+40%**

Error: `PlayerFightProgress.objects.select_related('fight').only('id').first()`

4.1 SQLAlchemy - Relationship Loading

```
# set children to load eagerly with a join
session.query(Parent).options(joinedload(Parent.children)).all()
```

Doc: https://docs.sqlalchemy.org/en/14/orm/loading_relationships.html

4.2 If you do not need ‘models objects’ - use .values()

```
obj.fight.boss.encounter_type
obj.fight.report.time_end_date
obj.fight.boss_id
obj.character.profession.description
obj.character.player_character.account

VS

.values(
   'fight__boss__encounter_type',                # fk__fk__field
   'fight__report__time_end_date',
   'fight__boss_id',                             # fk__fk_id
   'character__profession__description',
   'character__player_character__account',
   'character__profession_id',
)
```

4.3 Bonus: ORM serializers

```python
class FightsListSerializer(serializers.ModelSerializer):
    boss_title = serializers.CharField(source='boss__title')
    fight_end_time = serializers.DateTimeField(
        source='report__time_end_std',
        format=CoreConstant.DATE_TIME_FORMAT,
        default_timezone=pytz.UTC,
    )
    is_success = serializers.BooleanField(source='report__is_success')
    is_cm = serializers.BooleanField(source='report__is_cm')
    fight_duration = serializers.CharField(source='report__duration')
    boss_left_health_percent = serializers.CharField(source='report__left_health_percent')

    class Meta:
        model = Fight
        fields = (
            'id',
            'boss_title',
            'fight_end_time',
            'is_success',
            'is_cm',
            'fight_duration',
            'boss_left_health_percent',
        )
```

data = .values()

```python
class FightDetailSerializer(serializers.ModelSerializer):
    boss_title = serializers.CharField(source='boss.title')
    report_name = serializers.CharField(source='report.report_name')
    fight_end_time = serializers.DateTimeField(
        source='report.time_end_std',
        format=CoreConstant.DATE_TIME_FORMAT,
        default_timezone=pytz.UTC,
    )
    is_success = serializers.BooleanField(source='report.is_success')
    is_cm = serializers.BooleanField(source='report.is_cm')
    fight_duration = serializers.CharField(source='report.duration')
    boss_left_health_percent = serializers.CharField(source='report.left_health_percent')

    class Meta:
        model = Fight
        fields = (
            'id',
            'boss_title',
            'report_name',
            'fight_end_time',
            'is_success',
            'is_cm',
            'fight_duration',
            'boss_left_health_percent',
        )
```

data = object

5. QuerySet and ‘for-loop’

- avoid using DB requests inside ‘for-loop’
- .iterator() - direct access to fields, no RAM
- .values_list(‘field’, flat=True) - if you need only one field

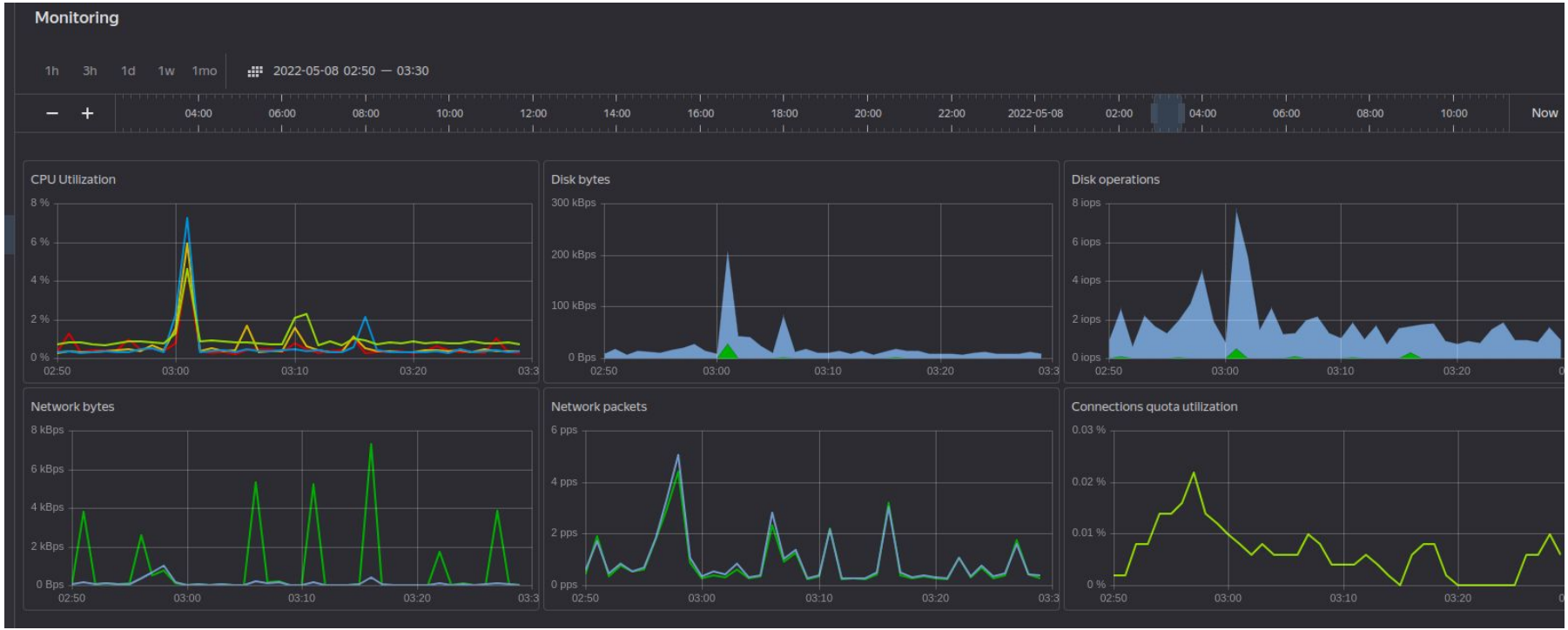
5.1 Before: DB request inside for-loop (daily task)

```
for benchmark in benchmarks:
    benchmark_id = benchmark.get('id')

    dps_1s_count = FindPlayerData.filter_1s_dps_for_benchmark(  # TODO: FIX !!!!
        do_count=True,
        build_obj_id=build_obj_id,
        boss_obj_id=boss_obj_id,
        encounter_mode=benchmark.get('encounter_mode'),
        specialization=benchmark.get('specialization'),
        player_role=benchmark.get('player_role'),
        benchmark_id=benchmark_id,
    )

    if dps_1s_count == 0:
```

5.1 Before: 2 tasks; ~30% CPU t1-2; t1 = 4 min, t2 = 12 min

! idle state - no new data !

5.1 After: 2 tasks; ~6% CPU t1; t1 = 2 min, t2 = 1 sec



! idle state - no new data !

5.2 Use .iterator() if RAM is limited or big data

```
some_values = Fight.objects.filter(
   query
).values(
   'boss_id',
   'report__duration_ms',
   'squad_damage_data__0__0__0',
).iterator()

for value in some_values:
   value.get('boss_id')
```

5.3 .values_list(‘field’, flat=True) if you need only one field

```
names = Character.objects.filter(
   player_character__account=account_name,
).values_list(
   'name',
   flat=True,
)
```

```
names with ‘flat=False’: [(‘name_1’,), (‘name_2’,), …]

names with ‘flat=True’ : [‘name_1’, ‘name_2’, …]
```

6. Django async

- first mentioned in 3.0
- async view, middleware, tests in 3.1
- async DB cache in 4.0 (not available in stable version)
- async QuerySet methods in 4.1 (august 2022)
- fully async in 4.2 LTS ???

6.1 Current cons of async in Django

- all requests are forced to be executed in one thread, i.e. sequentially;
- most of the middleware is synchronous;
- middleware can switch async/sync operation mode on the fly, but this leads to performance losses;
- and most importantly, because we work with API, - DRF does not support async.

A lot of tricks to make it work… Hello to FastAPi!

6.2 Dummy async not efficient!

```
@classmethod
@time_it('dude_retrieve()')
@async_to_sync
async def dude_retrieve(cls, dude_obj: Dude) -> dict:
    total_player_fights = await sync_to_async(FilterFights.count_fights_with_player)(dude_obj.gw2_account_name)
    from_6m_datetime = datetime.fromtimestamp(
        now().timestamp() - CoreConstant.SIX_MONTH_SEC,
        tz=pytz.UTC,
    )
    return {
        # general
        'most_played_classes': await cls.most_played_classes(dude_obj),
        'last_ten_logs': await cls.last_ten_logs(dude_obj),
        'stats_per_role': await cls.stats_per_role(dude_obj),
        'stats_per_encounter_type': await cls.stats_per_encounter_type(dude_obj),
        'stats_deaths': await cls.stats_deaths(dude_obj, total_player_fights, from_6m_datetime),
        'stats_downed': await cls.stats_downed(dude_obj, total_player_fights, from_6m_datetime),
        'stats_consumables': await cls.stats_consumables(dude_obj),
        'stats_mechanics_fails': await cls.stats_mechanics_fails(dude_obj),
        'stats_success_fail_per_day': await cls.stats_success_fail_per_day(dude_obj),
```

6.2 Pure async endpoint

```
async def dude_retrieve_async_view(request, *args, **kwargs):    # do DRF
    return JsonResponse(await DudeRetrieve.dude_retrieve_async(kwargs.get('dude_id', 0)))


@classmethod
async def dude_retrieve_async(cls, dude_id: int) -> dict:
    dude_obj = await sync_to_async(CRUDDude.get_dude_by_id)(dude_id=dude_id)
    # ...
    most_played_classes = await asyncio.create_task(          # avery method in separate task
        cls.most_played_classes(dude_obj)
    )
    # x9 methods with asyncio.create_task()
    return some_data


@staticmethod
@sync_to_async(thread_sensitive=False)                        # separate thread
def most_played_classes(dude_obj: Dude) -> list:
    pass
```

6.2 Result: from 23 to 68 RPS

6.3 Future of Django async

This adds an async-compatible set of methods and special methods to the `QuerySet` class (and, via the generic pass-through, Managers as well).

Included are:

- Async versions of all methods that do not return a QuerySet themselves, with an `a` prefix: `aiterator`, `aaggregate`, `acount`, `aget`, `acreate`, `abulk_create`, `abulk_update`, `aget_or_create`, `aupdate_or_create`, `aearliest`, `alatest`, `afirst`, `alast`, `ain_bulk`, `aupdate`, `adelete`, `aexists`, `acontains`, `aexplain`. Most are just wrappers around the underlying sync version, though some have a few performance shortcuts added.
- Async iterator ability on the `BaseIterable` that propagates through to all QuerySets as well as the results of `values()`, `values_list()` etc. It's not terribly efficient, but we can improve this progressively once we teach `compiler.results_iter` the wonders of async.
- A new `alist()` utility function for turning async iterables into lists asynchronously, because we do use `list()` quite a bit.

As a nice example, this means you can now write this kind of view:

```
async def myview(request):
    results = []
    async for row in TestModel.objects.filter(good=True):
        results.append(row)

    user = await TestModel.objects.aget(name=Andrew)

    return render(request, "index.html", {"results": results, "user": user})
```

7. Other techniques and suggestions

- .update() or .bulk_update() instead of .save() or ‘for-loop’ + .save() [*1]
- .bulk_create() instead of ‘for-loop’ + .create()
- m2m.add(*objs_list) instead of multiple .add()
- m2m.remove(*objs_list) instead of multiple .remove()
- do not call query methods on cached QuerySet [*2]
- use .annotate() or .aggregate() instead of ‘for-loop’ + some actions
- use cache

*1 .save() may me overridden

*2 think how to do 1 query, use .annotate()

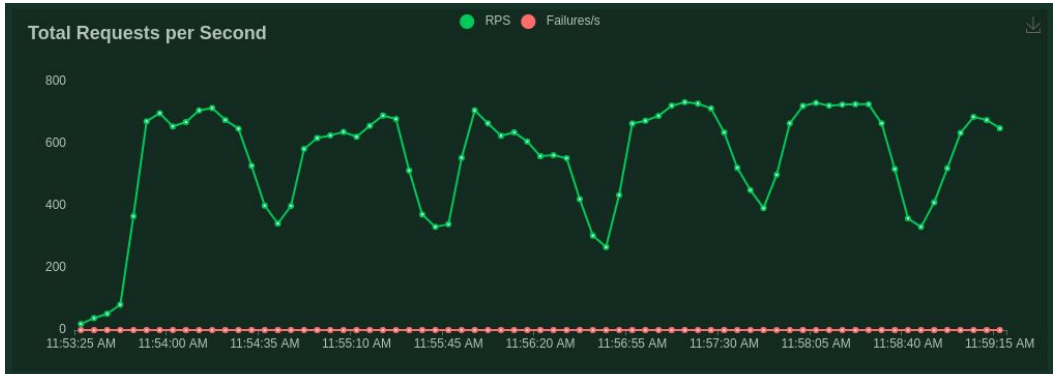
7. LocMemCache (-w4 gunicorn, 100 users)

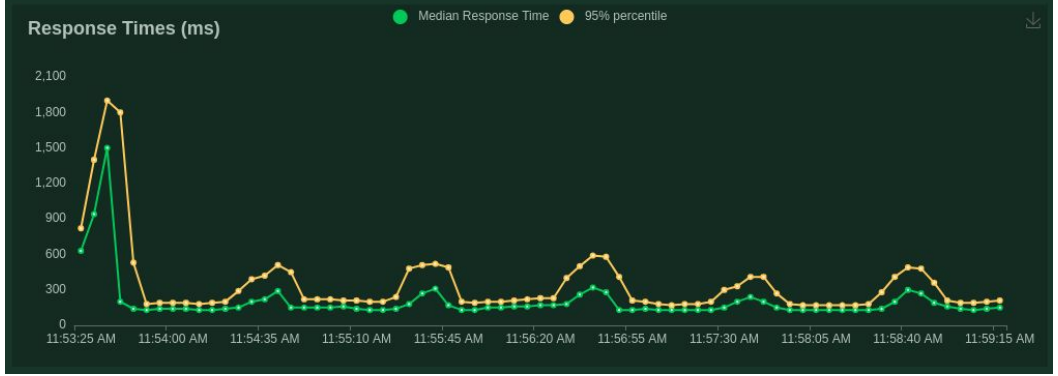
No cache:

- 25.6 RPS
- response 4.4 sec

With cache (1 min):

- avg 560 RPS
- avg response 240 ms





Useful links

- Performance and optimization - Django
- Database access optimization - Django
- QuerySet API reference - Django
- Cache framework - Django
- test project 1 django_10M_QS
- test project 2 FastApi_1k_qs
- Django Debug Toolbar
- Locust
- Aiohttp VS синхронные фреймворки - YouTube
- Querying Data, Loading Objects - SQLAlchemy
-